1986 Mid-Year Report

# EOS: A Project to Investigate the Design and Construction of Real-Time Distributed Embedded Operating Systems.



*Principal Investigator:* R. H. Campbell.

*Research Assistants:*
Ray B. Essick,
Judy Grass,
Gary Johnston,
Kevin Kenny,
Vince Russo.

*Software Systems Research Group*

University of Illinois at Urbana-Champaign
Department of Computer Science
1304 West Springfield Avenue
Urbana, Illinois 61801-2987
(217) 333-0215

**1986 Mid–Year Report**

# EOS: A Project to Investigate the Design and Construction of Real–Time Distributed Embedded Operating Systems.

*Principal Investigator:* R. H. Campbell.

*Research Assistants:*
Ray B. Essick,
Judy Grass,
Gary Johnston,
Kevin Kenny,
Vince Russo.

*Software Systems Research Group*

University of Illinois at Urbana–Champaign
Department of Computer Science
1304 West Springfield Avenue
Urbana, Illinois 61801–2987
(217) 333–0215

*ABSTRACT:* The EOS project is investigating the design and construction of a family of real–time distributed embedded operating systems for reliable, distributed aerospace applications. Using the real–time programming techniques developed in co–operation with NASA in earlier research, the project staff is building a kernel for a multiple processor networked system. The first six months of the grant included a study of scheduling in an object–oriented system, the design philosophy of the kernel, and the architectural overview of the operating system.

In this report, we will describe our operating system and kernel concepts. An environment for the experiments has been built and several of the key concepts of the system have been prototyped. The kernel and operating system is intended to support future experimental studies in multiprocessing, load–balancing, routing, software fault–tolerance, distributed data base design, and real–time processing.

TABLE OF CONTENTS



APPENDICIES

## 1. Summary.

This project is building an experimental operating system EOS, an example real–time embedded operating system for computer systems in aerospace applications. EOS is based on a distributed, object–oriented approach, and is specifically intended for distributed software applications in NASA's research and development program. The goals of this research include an investigation of the practical organization of kernels for multiple processor networked computers, real–time scheduling of tasks, the construction of system–based fault–tolerant support for distributed computing, and the design of basic service objects in a distributed, object–oriented operating system.

In the past six months, we have completed several studies:

- Completion of research into a high–level scheduling primitive for real–time systems called a Mediator. Judy Grass, the investigator, completed her Ph.D. which is appended to this report. Temporal logic was used to specify the semantics of the primitive. The primitive permits considerable flexibility in programming synchronization and scheduling while retaining modularity for that programming.
- Completion of the research on Path Pascal. The compiler for Path Pascal on UNIX was further debugged and the system stablized. The Path Pascal system has been used to support the operating systems class this Spring and Fall. Each class involved about 150 students using the system on an IBM S9000 microcomputer. Some of the students wrote projects in Path Pascal. The compiler has been distributed to some twenty sites.
- A study of device driver models. A device driver model was constructed for use with Path Pascal. It has since been adapted for use with EOS.
- Lessons learnt from Path Pascal in the construction of its run–time kernel have been applied to the construction of a prototype run–time process dispatcher for EOS. The dispatcher is coded in C++. Processes are programmed as instances of the class "thread". Classes are also used to program synchronization primitives.
- A bootstrap kernel has been constructed by removing all unessential detail from the UNIX System V kernel. The kernel provides a simple shell, access to disk, and linking and loading facilities and fits on a floppy diskette. The bootstrap kernel is being used to boot the EOS kernel. Utilities in the bootstrap kernel will be replaced by EOS utilities as they are being developed. (The C binding of C++ allows a simple replacement scheme.)
- A virtual memory scheme is being developed to share paged data amongst networked machines with the same data formats. Although this work is primarily useful for interconnecting workstations with faster processor servers, the technique also permits more flexibility by allowing virtual memory spaces to be multiprocessed.

- The structure of a lightweight RCP and message passing scheme has been devised which will permit exploitation of virtual circuit communication systems as well as packet switched systems. The structure also allows the underlying communication system to exploit networking capabilities and topologies to improve multicast and broadcast communications.
- The overall architecture of the operating system has been studied and consists of domains of objects and processes communicating through a lightweight remote procedure call. Within a domain, communications are optimized to procedure calls. Each domain can contain many lightweight processes, these are special instances of an "object". Objects can be shared concurrently by several domains. Processors are allocated to domains and they execute the active threads of control within those domains. Context switching between domains occurs because of priority concerns or because there are no threads ready to execute within the domain. Context switching between lightweight processes within a domain does not involve a major change in context. The file system, policy modules, user protection schemes and many other concerns of the operating system will be coded above the operating system kernel as system objects.
- The design of the kernel has been closely examined. The target is to produce the smallest kernel which will support the basic building blocks of the operating system. The kernel will support inter address space communication, task and process switching, and interrupt management.
- All kernel and operating system components will include real–time scheduling information and components. A short review of real–time systems suggested that such provisions should be built in to the system from the beginning, even if they are not used in the initial system prototyping.
- We have considered naming schemes for tasks, processes, and objects. A more detailed report will be produced later in the year.
- Protection will be provided by a hierarchical scheme that includes both access lists and capabilities. This scheme will be implemented above the kernel in the operating system. The kernel will only enforce protection provided by the hardware. The choice made here is to eliminate inefficiencies in the kernel which would be caused by repeated interpretation of access rights and to place the obligation for checking access rights with the object that requires protection (for example the file system.) The kernel will provide mechanisms to authenticate processes and sources and destinations of remote procedure calls.

The research project is using the tools and methodologies developed in earlier research[1] in co–operation with NASA [8,9]. The operating system overview is documented in [8,9,14,15,16,30,31]. The development environment supports implementation, reconfiguration, and testing of systems of component objects on both service–host and

[1] Some of the papers, reports, and theses that document our research are included in Appendix A.

stand–alone computers.

We intend to perform much of the operating system component testing on AT&T's 3B2/310, Motorola's M68020, Encore Multimax, VAX 750s, and on the AMETEK Hypercube. The kernel is being designed as a portable system and will be tested on several different architectures to ensure that machine dependencies have not been built into the software. The AMETEK cube will be used as a reconfigurable testbed for real–time systems. Each node of the cube can communicate by DMA to its five neighbors. The connectivity of the cube permits its use to model a hierarchical collection of shared buses or networks to which are attached groups of real–time processors. The availability of a large number of processors on the cube also enables some processors to be used for monitoring and simulating real–time I/O. A simulator for the cube exists on the VAX 750 and permits debugging.

Code will be mainly written in C++ and C to make it portable to all the machines that will be used in the project.

## 2. Mediated Objects

During the past year a design for *mediated objects* has been completed and formalized. The mediated object construct was developed to provide support for synchronization and scheduling for distributed systems programming. This support is essential to the development of complex real–time Embedded Operating Systems. Our interest in this topic comes from the observation that many existing tools for concurrent processing overly constrain concurrency, complicate scheduling and do not allow a modular approach to the specification of timing constraints.

Our interest in formal specifications have resulted in a complete temporal logic specification of the mediator construct. We believe that this specification is useful as unambiguous documentation of the design, as a guide to implementation and as an essential tool for program verification. This specification has also been a valuable check on the design and led to many improvements.

The design of the mediated object is presented in detail in a Ph. D. thesis presented in Appendix B. The informal presentation of the design has been the subject of a conference paper [31] which is in Appendix C. The temporal logic specification and the example of the use of the specification for verifying mediator programs are presented in the thesis.

The implementation of mediated objects should be a straight forward task. Few of the elements of mediators have not already been implemented in some form. The implementor has been provided with an unambiguous specification as a guide.

## 3. The Path Pascal Compiler

During the Spring semester, the compiler was once again used for the operating system class. A few bugs were found with the implementation and these have been corrected. The number of errors found was sufficiently small to consider the compiler

reasonably reliable. The VAX and SUN compilers have been distributed to a number of schools.

## 4. A Device Driver Model for Path Pascal

The Path Pascal programming language is designed to allow the user to experiment with the programming of multiprogramming systems. Its greatest use is in designing and simulating operating systems; for this purpose, however, the language itself is incomplete. One feature that the language lacks, by design, is any support for I/O devices. Appendix D contains a description of a device driver model for Path Pascal.

Devices are presented to Path Pascal as a set of external objects that can be linked with Path Pascal programs. Through these objects, the user can define a set of peripherals, such as disk drives and terminals, and allow the program to communicate with them. The model imposes a structure upon the communications with the device driver.

The model is currently being modified for use in EOS to provide actual device drivers.

## 5. Architecture of EOS.

EOS is built to support communicating real–time *tasks* that are composed of collections of *processes* performing operations on collections of *objects*. Each task runs in its own *virtual memory*. A task may use one or more processors to process its processes in which case the virtual memory is shared between the processors. Objects may be local to a task or process, shared between a group of tasks, or remote. The domain (accessible data) in which the task executes has four subdomains. Each subdomain is protected from the other subdomains by a hardware firewall. Individual objects and processes within the subdomains may be protected if the hardware supports segmentation and segmentation registers. Access to remote objects is accomplished through remote procedure calls and a *server* task. Other communication mechanisms are provided by additional communication objects.

The design of the multi–processor operating system will build on our experience and other related work. Of particular interest are the Mach [1] and Accent [50] systems and the V System [22,23,24]. The hardware model that we assume is a large number of high–performance processors (perhaps with vector operations) with local memory, interconnected through shared memory, high band–width networks or cross–bar switches. We believe that future networks appropriate for aerospace applications may involve optic fiber time division multiplexing, multiple frequencies or cross bar switching permitting efficient multicasts and virtual circuits. Such networks will be able to interconnect a large number of machines and provide a high bandwidth. The bandwidth of the network could make the CPU processors a network system bottleneck. Flexibility within the network configuration will permit experimental hardware to be attached to the networked computer system. Adaptability, reconfigurability, availability and reliability appear to be future important software considerations.

The design we have selected for EOS is based on the current evolution of operating systems towards object–orientation. This also reflects our experience with Path Pascal

based operating system design. The motivation for this evolution is that current operating systems are difficult to adapt to new, multi–computer/multi–processor architectures and reliable applications. The structure of many of today's operating systems are based on a sixties perception of time–sharing and real–time operation. Instead of a centralized timesharing or real–time operating system model, research is concentrating on a model in which operating systems are collections of independent but communicating objects. Such an approach enhances adaptability, reliability, reconfigurability and the ability to exploit multiprocessor hardware.

After a period of two decades, the timesharing system has begun to invade all corners of the commercial and industrial market. UNIX[2] is an example of one such successful system. During the two decades, knowledge about the structuring of computer systems has improved and new operating system concepts introduced. However, improved understanding of the structure of an operating system is not, by itself, a motivating force to develop a new operating system. It is our belief that the change in communication bandwidth will change how resources within a computer system are managed and that this will provide the major motivation to redesign operating systems. Adaptive and reliable system architectures provide additional motivation to explore new operating system designs. It is our intention to build a prototype embedded operating system which can be used as a testbed to decide how best aerospace operating systems for adaptive real–time applications should evolve.

However, our research is adopting a very pragmatic approach to the operating system structure. We have decided that we cannot impose overhead on the operation of the system that would not have been required in a normal time–sharing or real–time design. Using these two goals, our work has been to develop a new model for an operating system and kernel based on improved communications facilities and the studies that we and other researchers have made into the structures of operating systems.

At the lowest level of the operating system is a *nugget*. The nugget supports *task and process switching* and *synchronization* between processes. The nugget is written in C++ and a small amount of assembler code. A prototype nugget is listed in Appendix E. Upon the nugget is built a kernel. The kernel provides a user level system interface. It is through the kernel interface that tasks may request other operating system services.

The *kernel* will be constructed as a set of co–operating objects written in C++. Using our experience from LINK, these objects will provide a standardized set of functions which may be used to build a variety of different operating system service interfaces to the user. The functions are organized into layers (for example, remote procedure calls will be mapped onto sequenced messages which will in turn be mapped onto asynchronous message passing primitives.) The interface between the kernel and the user is modeled after the object–oriented notion of a *class*. The interface takes the form of a set of primitives we have nicknamed a RUSK or reduced universal system kernel. The primitives will include process creation, deletion and various communication primitives like open, read, write, and close. The parameters to these primitives are chosen so

[2] UNIX is a trademark of AT&T.

that name resolution issues are encapsulated within the kernel. (In UNIX, the physical file descriptors of open files and process identifiers of active processes are bound to a single processor. This prevents UNIX United and its implementations from allowing explicit reference to remote running processes and open file descriptors.)

*Extensions to the kernel* take the form of additional layers which specialize the kernel interface. Kernel extensions include real–time, fault–tolerance, distributed computing functions like load balancing, and other specialized functions. These kernel extensions are integrated with the kernel interface using a model based on the object–oriented notion of a *subclass*. Users may add kernel extensions in a similar manner to the way in which UNIX System V permits filters to be pushed onto a stream. Tasks may select the appropriate kernel subclass at task creation time.

When the processes of a task request access to a non–local object, the kernel binds that request to a particular *shared* or *remote* object using name–server objects provided by the operating system. Local objects may be accessed by means of procedure calls. Bound, shared objects may be accessed by means of "gated" procedure calls. The kernel system call interface is also implemented as a "gated" procedure call. Bound, remote objects may be accessed by means of local stub objects and remote procedure calls made through the kernel interface.

The operating system consists of an operating system task and a library of objects. It can be organized as a distributed operating system if required to overcome the possible limitations imposed on the software by a particular parallel architecture. For example, on a hypercube architecture, processors within the cube may not have access to disk controllers. For such systems, it would be more efficient to place the physical file system operating system objects on nodes at the periphery of the cube that do have access to disk controllers. The reorganization is performed by replacing the objects with stubs that make low level remote procedure calls to the file system remote objects on specific processors.

We intend to perform much of the operating system component testing on AT&T 3b2, M68020 computer systems, VAX 750s, and on the AMETEK Hypercube. The kernel is being designed as a portable system and will be tested on several different architectures to ensure that machine dependencies have not been built into the software.

The resulting code will have many experimental research applications. We hope to use the AMETEK cube implementation as a reconfigurable testbed for real–time systems. The structure of a typical real–time control task is shown in Figure 4.1. A hierarchy of such tasks might be used to implement a real–time application. The connectivity of the cube permits its use to model a hierarchical collection of shared buses or networks to which are attached groups of real–time processors. In such a hierarchy, real–time processes that interact with sensors and actuators with fast responses are placed at the leaves of the hierarchy on independent processors served by highly available buses. Status information is gathered from and control information is passed to these processors by one or more control processors which reside at the next level on the bus hierarchy. In turn, these control processors communicate to higher levels of control over the next level of the hierarchy. The availability of a large number of processors on the cube

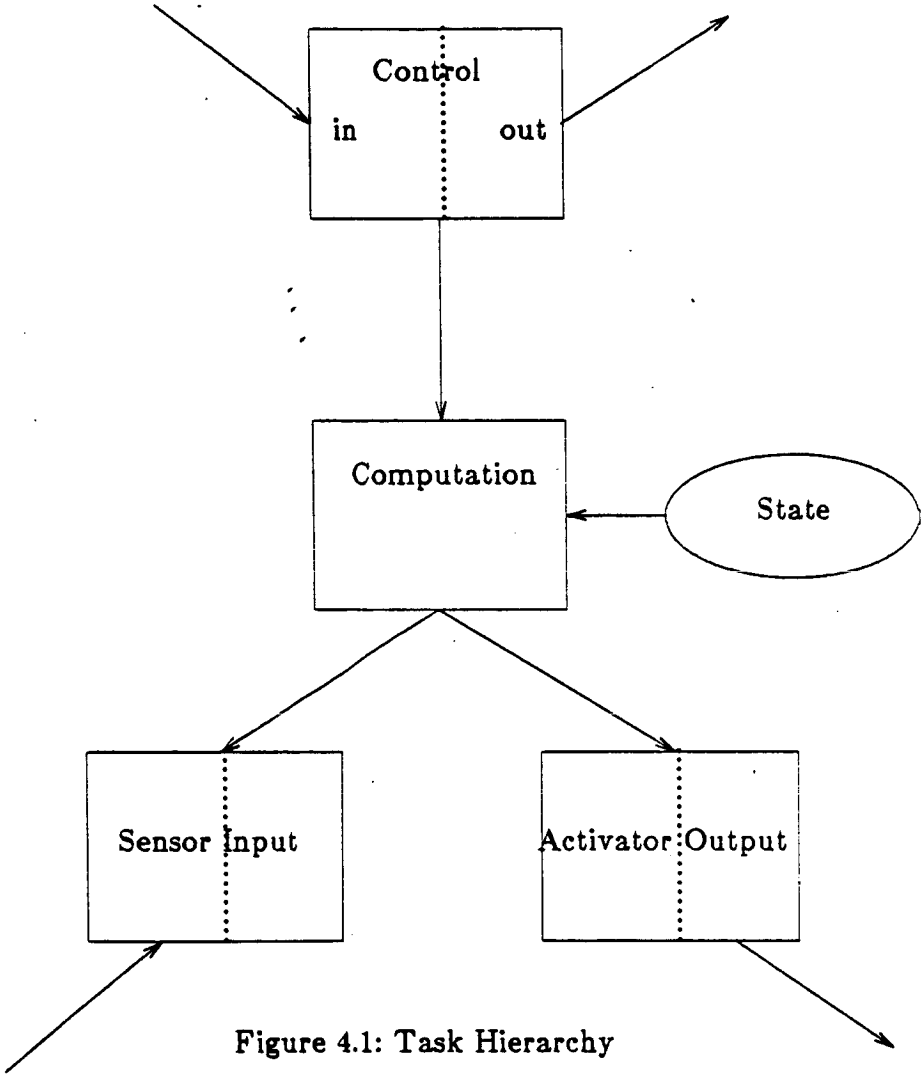



Figure 4.1: Task Hierarchy

also enables some processors to be used for monitoring and simulating real–time I/O.

The architecture of EOS may change as we develop a more concrete implementation. Issues concerning the virtual memory organization of the system could still alter the design. Also, we hope to gain feedback from various other NASA investigators about our proposed architecture.

**6. Communications.**

There have been many communication primitives proposed for distributed systems. In EOS, we intend to support well those primitives for which there is appropriate hardware support. To this end we provide several high–level communication primitives like the remote procedure call and interfaces to allow other communication mechanisms to be built efficiently.

### 6.1. Overview

The overall underlying architecture of inter–process and inter–processor communications for the EOS kernel is one of lightweight asynchronous message passing; the closest analogue to it among existing systems is the V Kernel of Cheriton [22,23,24]. It allows for rapid message–based communications between processes, whether at the same node or at different nodes, provides a multicast facility for communicating with groups of processes, and allows for real–time scheduling of message traffic. In order to be as lightweight as possible, it is set up to minimize the amount of copying required in the processing of a message. A remote procedure call is built on top of the message passing mechanism for use by the object operation invocation scheme of the system.

The communications architecture comprises at least five distinct areas that have been identified so far:

(1) A *buffer manager* to handle the allocation of buffer space for communications traffic; this service must be centralized to avoid needing to copy messages between layers of the system.

(2) A set of *communication device drivers* to handle the details of interfacing for the communications hardware present on a particular system.

(3) An *asynchronous message service* to control the switching of messages among the processes and nodes of the system.

(4) A *router* to choose the routing for a message to follow. The router must be able to choose routings for multicasts, and to provide reliable estimates of the real–time requirements for getting a message to its destination.

(5) A *sequencer* to ensure (if necessary) that messages arrive in the order in which they were sent, and to guarantee once–only message transmission.

Remote procedure calls, remote object servers, and the like are implemented atop this basic framework (Figure 5.1).

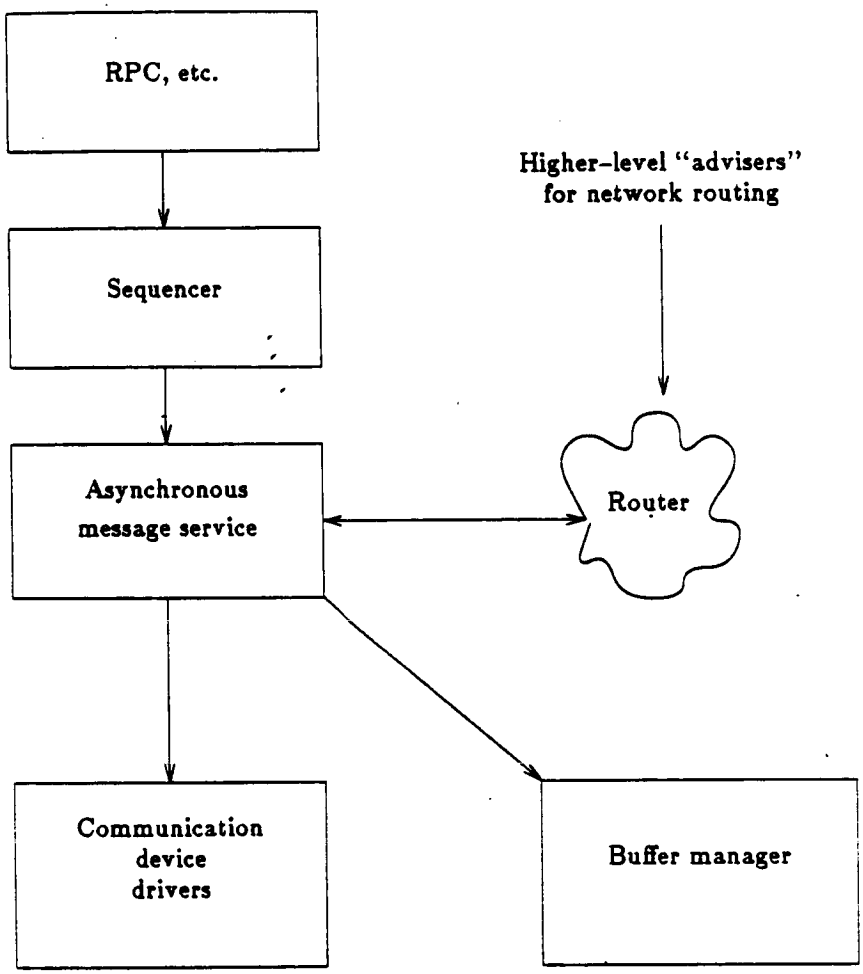



**Figure 5.1. Communications overview**

### 6.2. Buffer manager.

The buffer manager provides a single point where message buffers can be allocated. The address space in which they are allocated remains visible throughout the kernel, and hence the messages need not be copied between the layers of the communications facility.

Each buffer comprises an area of memory of a specified size, and a usage count. The usage count is maintained by the buffer's "use" and "free" functions, and keeps track of the number of active references to the buffer. When the usage count reaches zero, the buffer may be reused by another operation.

The rationale behind maintaining a usage count, rather than simply providing Unix-like "allocate" and "free" operators, relates to the fact that a multicast message

may need to be forwarded over several communication links. Maintaining a usage count seems the easiest way to handle this situation.

### 6.2.1. The `buffer-manager` object.

The `buffer-manager` object is the global entry to the buffer manager. It provides a single entry point:

```
buff = buffer-manager.allocate (size)
```

The `size` argument is the size of the buffer to allocate, in bytes. The `buff` return is a pointer to a `buffer` object containing a buffer of the specified size and a usage count of 1.

### 6.2.2. The `buffer` object.

A `buffer` object represents a buffer of memory, its size, and its usage count. It provides four basic operations:

```
siz = buffer.size
```

The size of the buffer (specified when the buffer was allocated) is returned as `siz`.

```
ptr = buffer.area
```

A pointer to the memory area allocated for the buffer is returned as `ptr`.

```
buffer.use
```

The usage count of the buffer is increased by one.

```
buffer.free
```

The usage count of the buffer is decreased by one. If the usage count has reached zero, the memory area associated with the buffer is released. Any use of this function that may cause the usage count to reach zero must be regarded as spoiling the buffer pointer; the pointer must not be used again after this function returns.

## 6.3. Communication device drivers.

The communication device drivers handle the physical hardware interface to the communication links among the nodes of the system. They are implemented using the "physical device object" model that has already been proposed for Path Pascal. Such an object has three identified functions:

```
device.initialize (data)
```

Initializes the device for communications. The `data` argument is an implementation-dependent structure that describes the configuration of the device. The `initialize` operation places the device in a quiescent state, ready for the remaining operations to use it.

```
status = device.operate (command, buffer)
```

Performs an operation on the device, as specified by `command`. The available commands include at least `read` and `write` operations. The buffer is read from or written to the device according to the command. The requesting process is delayed until the operation

is completed; the `status` return then reflects the success or failure of the operation.

`status = device.await-interrupt`

Delays the requesting process until an urgent condition (*e.g.*, an unrequested write at the other end of a communication link) is detected. When such a condition is detected, a description of the situation is returned as `status`.

The device driver routines will be located in the operating system, a virtual memory and domain set aside for operating system data and tasks. The device drivers have access to the real memory locations required to perform their task within their virtual memory. Short interrupt handling routines will interface particular device drivers into this scheme. We believe that context switching time should be sufficient to allow the device drivers to be placed in virtual memory. However, if not, they can be moved into the kernel.

### 6.4. Router.

The object structure of the message router has yet to be fully elaborated; hence its appearance in Figure 5.1 as a "cloud". Its basic functionality is that, given a list of nodes to which a message is directed, it must determine some nearly–optimal routing by which the message can be sent to those nodes. The routing chosen may depend on the urgency of a message (urgent messages my need to get minimum–time routings; non–urgent ones, minimum–cost routings) and hence the router must be aware of the priority of a message for which it is finding a routing.

Since the system is directed towards real–time operations, the router must also have some knowledge of expected communication delays in the network, and be able to estimate the expected elapsed time for a message to reach its destination. For messages with deadline schedules, it may be required to return a denial indication if its estimates indicate that a message is unlikely to reach its destination in the required time.

In order to minimize processing when a message must be forwarded by one or more intermediate nodes before reaching its destination, it may need to provide out–of–band information directed at the router on an intermediate node; its interface with the asynchronous message manager must provide for this.

If an attempted transmission of a message fails, the router must be informed, in order that its network tables may be updated to reflect the status of the node or communication link that failed.

Higher levels of the system may require an interface to the router in order to provide it with "advice" relating to the choice of routes (for instance, data on congestion of the communication pathways).

One possible set of interfaces to the router would look like:

`route-set = find-routing (buffer, dests, info)`

Finds a routing to get the message in `buffer` to the set of node–process pairs specified by `dests`. The `info` argument is a data structure providing information for the router to make its decisions; it must at least include specifications for the urgency and deadline

of the message. The `find-routing` request returns a set of ordered pairs `(link, data)` where `link` identifies a communications link on which the message is to proceed and `data` is a buffer containing out-of-band data to be transmitted with the message in order to advise the router at the other end of the link how to proceed.

```
forward (buffer, data)
```

Find a routing for the next hop for a message which has arrived from another node. The `buffer` contains the message; the `data` argument is another buffer containing the out-of-band data supplied by the router at the previous node. Just as with `find-routing`, a set of `(link, data)` pairs is returned.

```
fail (link-info)
```

Record that a particular link or node has failed. The `link-info` argument gives data on the particular component that has failed and the type of failure, in order that the router can be advised for its decisions relating to future messages.

Research is in progress on further specifying these interfaces, and on heuristics for choosing message routings on the hypercube configuration. Preliminary results on the latter topic will be reported in a forthcoming paper.

### 6.5. Asynchronous message manager.

The asynchronous message manager provides a facility, similar to that in the V System, for reliable delivery of messages addressed to some set of processes in the network. Full end-to-end acknowledgement of messages is provided, to ensure "at least once" semantics for delivery of the messages. If "exactly once" and "delivery in sequence" semantics are required, a higher-level sequencing layer can provide them.

#### 6.5.1. Functions.

At least four functions are provided: "send,", "send and await reply," "inquire," and "receive." A "message" object is implemented to describe messages in progress.

```
message = send (process-list, buffer, info)
```

Sends the message contained in `buffer` to the set of (node, process) pairs specified by `process-list`. The `info`argument specified other information about the message, including at least its priority and deadline for completion.

```
(buffer, status) = send-await-reply
                   (process-list, buffer, info)
```

Sends the message contained in `buffer` to `process-list`, exactly as with `send`. The requesting process is then blocked until one of the processes in `process-list` sends a reply message to it. When the reply is received, it is returned as a (buffer, status) pair, exactly as with `reply` below. The `status` return must be able to specify at least:

- Failure of the target process.
- Failure of the target node.
- Network failure; failure of enough links to make the target node unreachable.

- Inability to complete the request before its deadline.

`status = inquire (message)`

Inquires as to the status of the specified message (as returned by `send`). The requesting process is not blocked. The status return includes at least the four possibilities specified above, plus an "in progress" status which must also provide an estimate of the completion time for the request.

`(buffer, info) = receive (control)`

Receive the next message destined for the requesting process. The message text is returned in `buffer`; the `info` return gives data on at least the identity of the sending node and process, the original destinations for the message, and the requested completion time. If no message is pending, the requesting process is optionally (by an indicator in the `control` argument) blocked until a message is received.

### 6.5.2. Comments.

When a message arrives over a link, the first thing that the message service must do is to determine whether it is addressed to a local process, and if so, queue it accordingly. If it is destined for a remote process, the message service must also call the router to obtain forwarding information, and pass the message along over the next link.

Any incoming message for which the corresponding process has failed must be rejected with an appropriate status message returned to the sender. If the message was sent using the asynchronous `send` operation, the rejection will be detected by the next `inquire` operation.

All link operations must be provided with a timeout capability to detect failed nodes and links.

### 6.6. Sequencer.

The sequencer is the next layer above the asynchronous message manager. It provides exactly the same set of functions as the message manager, but in addition ensures that duplicate messages are weeded out (it is possible for a single message to arrive at a node over different routes in the lower–level layer) and that messages are received in the order in which they are sent. It provides the infrastructure needed to implement a general remote–procedure–call protocol.

## 7. Virtual Memory.

The hardware we are targeting the system for is presumed to be a segmented, paged architecture similar to the IBM 360–370. However, we assume a large virtual memory address and a large (2–32k) segment address space or better. Pages are expected to be approximately 2k. In the initial design we will use a 3b2–like architecture as it most closely resembles an "standard" architecture with common features to many existing machines. The 3b2 architecture has a 32 bit virtual memory address and uses 4 segment tables per virtual memory (2 bits of the address), each with 8k segments (13 bits of virtual address) divided into 64, 2k pages (11 bits of virtual address).

Hardware support for separate virtual address spaces will be used at this point only to protect the virtual address spaces of tasks/objects from one another. The use of virtual memory to support address spaces larger than physical memory, via the use of secondary storage, will not be implemented at this time (although its eventual addition will certainly be kept in mind during the design process). We feel that this is a valid initial simplification for the prototype in that the types of systems targeted for support by this project are those which are primarily of the nature of embedded control systems where huge address spaces are not crucial.

Essentially, we intend to use the segmented virtual memory to provide protection for objects belonging to the kernel, operating system, and specific user application packages. For example, the file system may be implemented as an object which is shared by applications. It will reside in virtual memory and be protected from misuse by the application. Different file systems may coexist in the system. (This is why we choose to place the file system in virtual memory as a separate object rather than as part of the kernel.) However, we recognize the problems of making each object in the system an individual "segment" in a strictly linear segmented, paged address space which does not have segment registers. Such a scheme might impose a large penalty, (but perhaps small in terms of each segment access), on cross–domain procedure and object operation invocations and require difficult to generate relocatable code.

Our proposed scheme partitions the address space into *four sections*; an *address space for the nugget and kernel* to be shared by all virtual memories; an *address space for a collection of sharable objects* (including class code and instances); an *address space for task information* which would include data and text for the objects and lightweight processes owned and used locally by the task; and an *address space for a collection of lightweight processes* containing local process variables, parameters, and stack (see Figure 6.1.)

The nugget and kernel section and the task section are permanent members of the virtual memory and remain unchanged except for the acquisition and disposal of additional segments and pages required by the task. The object section may be exchanged on demand in order to access a different collection of objects with different protection requirements. The objects mapped by this section may be shared between groups of tasks. The process collection section may also be exchanged on demand for another process collection. Process collections are not shared between tasks however. The section mechanism provides a large grain protection mechanism that prevents different collections of lightweight processes from interfering with one another.

The collection of objects that constitutes a section can be stored on permanent file storage as persistent objects. When a task first requests the use of a collection of objects, it is loaded as a binary image into real memory. During the course of the computation, parts of an object may be paged out to backing store as space requirements dictate. Replacing one object section with another object section requires the replacing of the segment table of the objects with the segment table of the other objects. On a machine like the 3b2, this involves changing one pointer. On a machine like an Intel 286, this involves changing some, but not all, segment registers. At the end of the task's

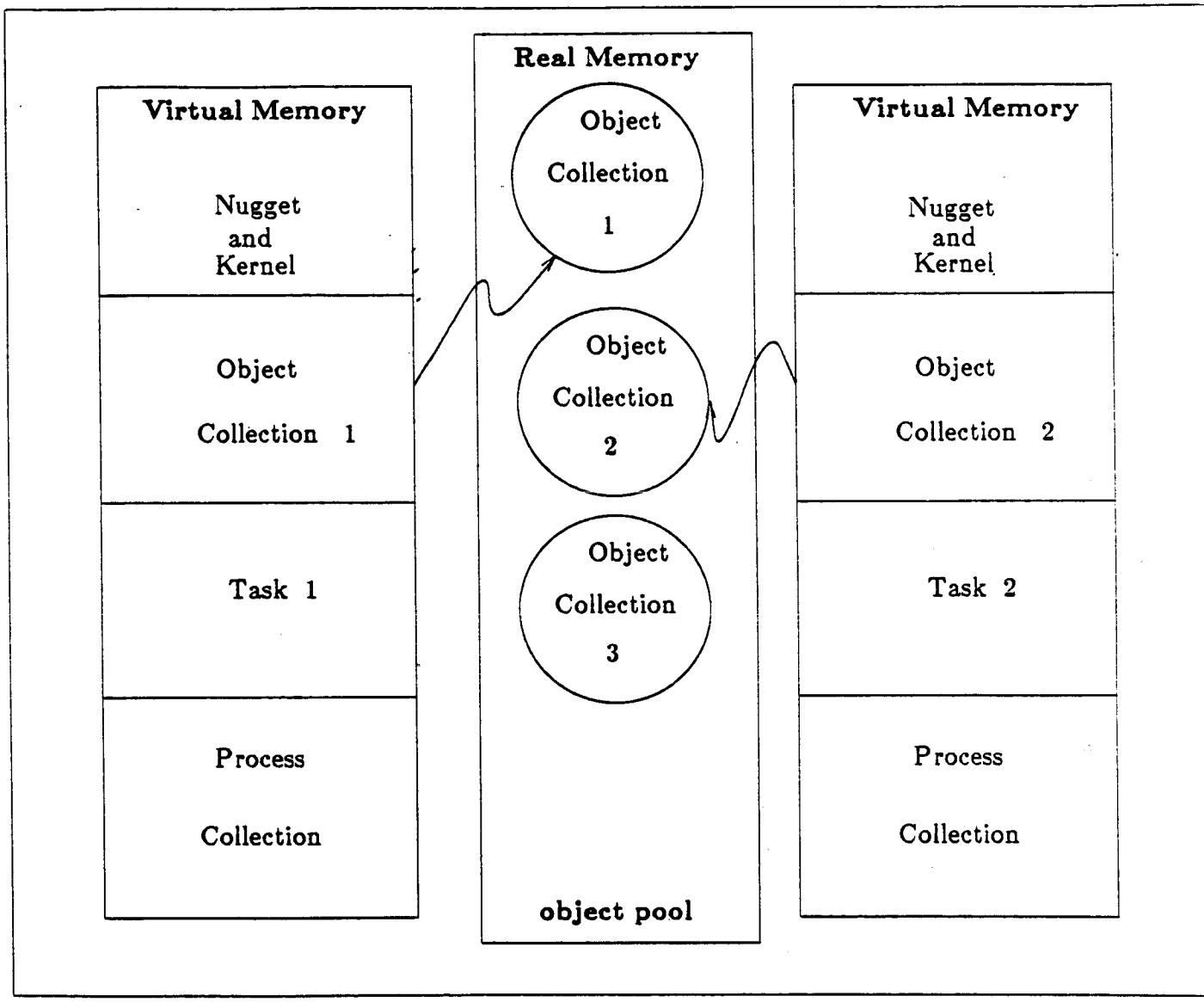



Figure 6.1: Virtual memory layout showing mapping of sharable objects

use of the object, its binary image may be stored back on permanent storage.

A collection of objects may be shared between different concurrent tasks by mapping the collection into the object section of more than one task. Synchronization of the operations performed on an individual object is the responsibility of the developer of that object. A usage count ensures that a collection of objects is not stored back on permanent storage until the number of tasks accessing that collection drops to zero.

Since each collection of objects (processes) use the same linear address space; two collections of objects (processes) that have already been mapped into a binary image

cannot be loaded into the same object (process) section. When an operation on an object is invoked, the nugget and kernel must ensure the correct updating of the object section. For example, an object may request an operation on an object in a different collection. This will result in a change of the object section. Similarly, when a return from an inter-object invocation is made, the nugget and kernel must ensure that the old object section is restored before execution is resumed.

A task may have the need for several different process collections to implement its application. For example, a real-time task might be composed of several collections of processes, each controlling a separate real-time experiment. It is possible to map different collections of processes into the process section at different times, allowing collections of lightweight processes to interact efficiently within the same address space while physically protecting other collections of processes of the task from possible harmful access. The task schedules its lightweight processes through operations on the thread objects (although the processes are dispatched by the nugget.)

Rather than relinquish control of the CPU when a lightweight process blocks, a task may choose to reschedule a different lightweight process. This mechanism allows the nugget to dispatch a process that has the same virtual memory requirements as the previous process. Within the operating system, a different level of scheduling dictates when a task must relinquish control of the CPU.

An invocation of an operation on a shared object is achieved by a "gated" procedure call. If the object is resident in the object section, this amounts to a direct procedure call to the operation code. If the object is not present, the invocation results in a trap to the kernel.

For various implementation and efficiency reasons, the protection mechanism for objects is not as secure as might be desired unless the hardware has segmentation registers. For example, the process stacks are kept in a shared section of address space. Complete protection from undesirable effects of an invocation of an object can only be achieved by placing that object in a separate virtual memory space with a task that operates as a server for the operations on that object. The object is then accessed through the remote procedure call communication mechanism. Stubs for remote objects are loaded into the address space of the task where they can be shared by all the processes of the task. A remote procedure call transmits a message to the server task for the particular object. The server task implements the remote procedure call by using a lightweight process to execute the operation on the object.

An extension to the virtual memory scheme allows the processes of a task to be executed on several processors provided that the hardware uses similar data representations. In a scheme devised by Essick for UNIX but adapted for EOS, the data segments of virtual memory of a task may be paged between two or more processors. The code segments that implement the processes are partitioned and compiled for each participating processor. Paging of data segments occurs on demand and it is used to synchronize the updating of and access to shared data. The kernel and nugget section of the virtual memory resides only in one machine. The scheme permits a task to reside on both an I/O processor and a high-speed processor. The two processors are closely coupled

through the use of a common virtual memory. When used in this way, the scheme allows I/O driven computations to use the speed of a high–speed processor transparently, that is without modification of the code. Another use of the same scheme is to provide load balancing by off–loading parts of a user task onto other processors dynamically as the computation progresses and load changes on individual processors. A detailed proposal for this scheme is in preparation as part of Essick's Preliminary Ph.D. thesis proposal.

## 8. Tasks and Processes

A task has an associated viftual address space in which processes are (potentially) executing. The virtual address space of one task offers protection from interference from other tasks. The resources of a task will include the address space itself and (usually) access to services provided by other tasks.

The task's processes are the main object of activity in the proposed system. Processes in a task can provide services requested of it *by* other tasks, and can actively initiate requests for service *to* other tasks.

Tasks and processes can be dynamically created and destroyed and will provide the basis for the support of "pluggable service modules". For example, there may be a "memory allocator service task" or a "communications services task". In fact, there may be more than one type of server for a given resource providing different operations and/or different semantics for the same operations. For example, there might be a "synchronous communications service task" and also an "asynchronous communications service task".

To meet real–time objectives, a task should be able to specify the scheduling of its processes using an approach similar to that of the Mediator [30,31]. Also, context switching between processes within a task should be very fast. In this regard, the processes could be considered to be "lightweight," with the major portion of the state information common to all processes within the task (and, therefore, not needing to be saved or restored during an intra–task context switch). Figure 7.1 shows two tasks which contain many lightweight processes. Context switching between processes in user space in different tasks will require changing the virtual memory mapping and hence will not be lightweight. However, context switching between processes within the kernel will be lightweight, the virtual memory mapping is only changed on a need to access user space basis.

A prototype scheme for dispatching lightweight processes has been programmed in C++ and is shown in Appendix E.

## 9. Object Support.

The system proposed is an object–oriented system. This is important in that it promotes the encapsulation of data and state information of system entities. It is especially important for fault–tolerant systems in that periodic "checkpointing" of critical objects is straightforward in such a system. In addition, this encapsulation allows much easier migration of entities to other nodes. The prime factor in determining the

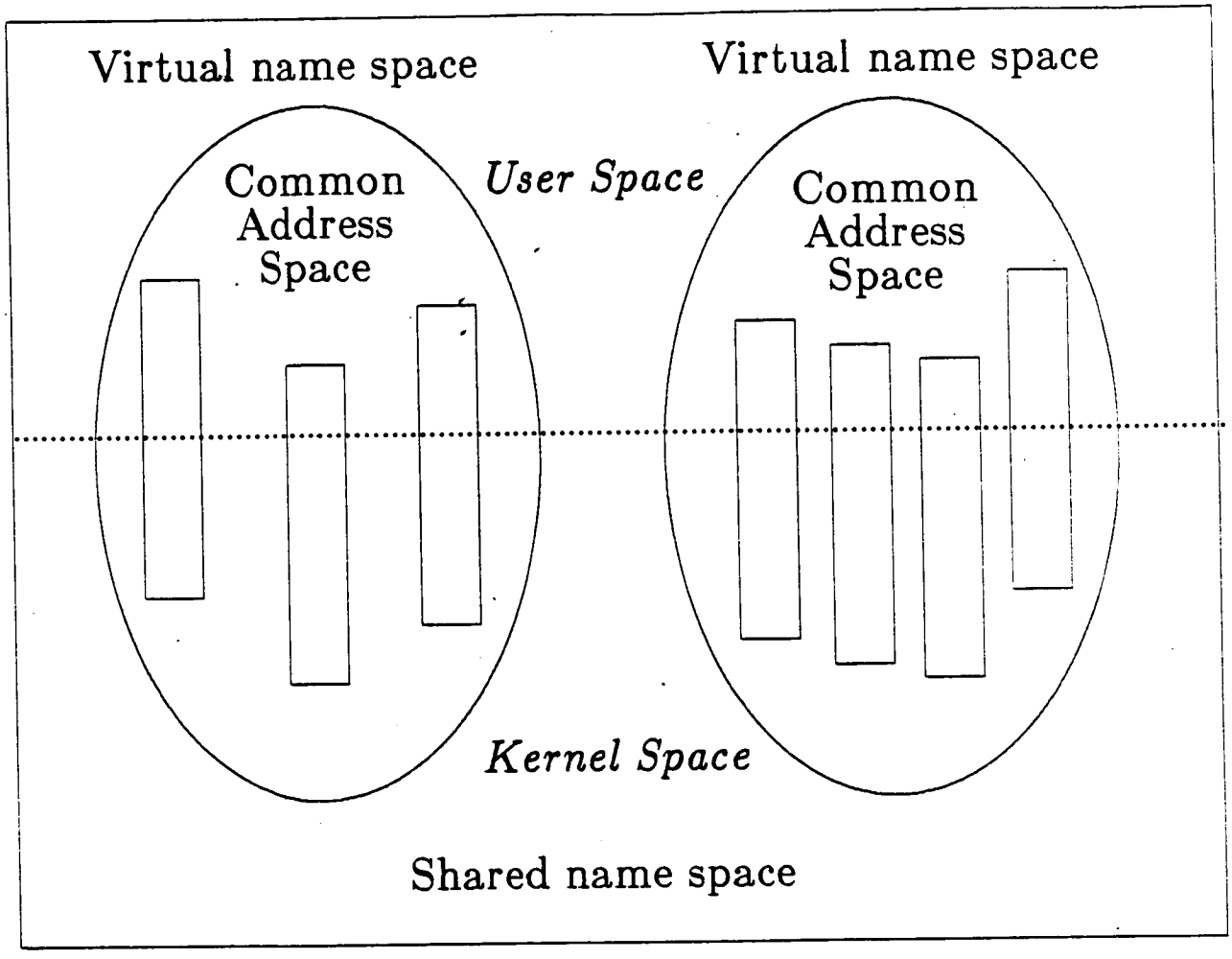



Figure 7.1: Tasks and Processes in name spaces

implementation strategy is to provide object–orientation without undue real–time overhead.

A task allows one or more processes to access one or more objects within a virtual memory. Each object has a set of operations and belongs to a class or subclass. An object may reside in the local address space of a task and its lightweight processes or it may reside in a global system address space and be shared between between tasks. To permit an object to be accessed remotely, a service task provides lightweight server processes which receive requests and invoke operations on the object. A client process accesses a remote object by executing a call to a stub object which is created within the task. The stub object within the task communicates through the nugget and packages a remote procedure request across the network. This remote procedure request is received

by the server task and given to a particular lightweight process server.

The C++ notion of a class and subclass is very powerful. Generic classes allow families of classes to be created conveniently. In EOS, a *thread* class supports the notion of a lightweight process. Prototype C++ classes used by a task to provide the facility of creating and destroying processes are shown in Appendix E. Process scheduling and synchronization primitives are also provided by the C++ class approach. Appendix E contains a class providing P, V operations. Current work is devising monitor and guarded command *generic* C++ classes. In addition, classes may be used to provide simple communication channels. For example, an extension of the queue class in Appendix E that uses semaphores to control access to the queueing operations provides a FIFO synchronization discipline on producer/consumer processes.

## 10. IPL

A UNIX–based IPL program has been built to allow fast prototyping and debugging of the EOS nugget and kernel code. The IPL program includes a simple UNIX shell and permits the loading and execution of a single binary image. The IPL system is small enough to be stored on a single floppy. The current IPL program is available for the 3B2.

## 11. Conclusions

The project has accomplished much in the past six months. Devising an organization for EOS has taken a longer time than we had first planned, however the structure we now have for the operating system makes an object–oriented operating system possible to build without imposing undue penalties on performance.

In the next few months, we shall be contacting other NASA contractors and researchers with our operating system proposals for comment. During this time, we shall proceed to build a prototype system. We hope that the framework we have devised can be used as the foundations for experimental studies of many aspects of object–oriented real–time operating system design for aerospace applications.

## 12. References


1. Accetta, Mike, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, "Mach: A New Kernel Foundation for UNIX Development," Computer Science Department, Carnegie Mellon University, 1985.
2. Allchin James E. and Martin S. McKendry, "Synchronization and recovery of actions," In: *Proc. 2nd Principles of Distributed Computing,* Montreal, P.Q., Canada, 1983.
3. ——. "Support for Objects and Actions in Clouds," Technical Report GIT–ICS–83/11, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia, 1983, p. 24.
4. Allchin, James E., Martin S. McKendry and William C. Thibault. "Clouds: A Testbed for Experimentation in Distributed Systems", Working Paper 2, Interprocess Communications, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia, 1982, p. 23.
5. ——. "Clouds: A Testbed for Experimentation in Distributed Systems", Working Paper 3, Status Report, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia, 1982, p. 23.
6. Andrews, Gregory R., "Synchronizing Resources,", *ACM Transactions on Programming Languages and Systems* (October 1981) vol. 3, no. 4, pp. 405–430.
7. Birrell, Andrew and Bruce Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems* (February 1984) vol. 2, no. 1.
8. Birman, Kenneth P., Thomas A. Joseph, Thomas Raeuchle, and Amr El Abbadi, "Implementing Fault–Tolerant Distributed Objects," *IEEE Transactions on Software Engineering,* vol. 11, no. 6., Jun 1985.
9. Black, Andrew P., "Supporting Distributed Applications: Experience with Eden,"In: Proceedings of the Tenth ACM Symposium on Operating Systems Principles, 1985.
10. Campbell, Roy H., Jeff Donnelly, Raymond B. Essick, Judith Grass, Dirk Grunwald, Pankaj Jalote and David A. McNabb. "The Embedded Operating System Project," 1984 Mid–Year Report, NASA GRANT NSG 1471, Software Systems Research Group, Department of Computer Science, University of Illinois at Urbana–Champaign, Urbana, Illinois, 1984.
11. Campbell, Roy H., Raymond B. Essick, Judith Grass, Dirk Grunwald, Pankaj Jalote, Kevin Kenny and David A. McNabb. "The Embedded Operating System Project," 1985 Mid–Year Report, NASA Grant NSG 1471, Software Systems Research Group, Department of Computer Science, University of Illinois at Urbana–Champaign, Urbana, Illinois, 1985.

12. Campbell, R. H., K. Horton, and G. G. Belford, *Simulations of a Fault–Tolerant Deadline Mechanism,* **Digest of Papers FTCS–9: Ninth Annual International Symposium on Fault–Tolerant Computing**, Madison WI, June, 1979, 95–102.

13. Campbell, R. H. and R. B. Kolstad, *Path Expressions in Pascal,* **Proceedings of the Fourth International Conference on Software Engineering**, Munich, September 17–19, 1979, 212–219.

14. Campbell, R. H. and R. B. Kolstad, *Practical Applications of Path Expressions to Systems Programming,* **ACM79**, Detroit, 1979, 81–87.

15. Campbell, R. H. and R. B. Kolstad, *An Overview of Path Pascal's Design,* **Sigplan Notices**, Vol. 15, No. 9, pp. 13–14, September, 1980.

16. Kolstad, R. B. and R. H. Campbell, *Path Pascal User Manual,* **Sigplan Notices**, Vol. 15, No. 9, pp. 15–24, September, 1980.

17. Kolstad, R. B. and R. H. Campbell, *Directions for User Defined Communication for Distributed Software,* **Proceedings of The International Conference on Parallel Processing**, IEEE 80CH1569–3, pp. 188–189, Boyne MI, August 26–29, 1980.

18. Campbell, R. H., *Distributed Path Pascal,* **In Distributed Computing Systems**, (Editor Y. Paker and J.–P. Verjus), Academic Press, 1983, pp. 191–224.

19. Campbell R. H. and B. Randell, **Error Recovery in Asynchronous Systems, IEEE Transactions on Software Engineering, August,** 1986.

20. Campbell, R. H., C. S. Beckman, L. Benzinger, G. Beshers, D. Hammerslag, J. Kimball, P. A. Kirslis, H. Render, P. Richards, R. Terwilliger, "SAGA: A Project to Automate the Management of Software Production Systems," 1985 Mid–Year Report, NASA Grant NAG 1–138, Department of Computer Science, University of Illinois, Urbana, IL.

21. Campbell, R. H. and T. Anderson, *Practical Fault Tolerant Software for Asynchronous Systems,* **SAFECOMP 83, Third International IFAC Workshop on Achieving Safe Real–time Computer Systems**, Pergamon Press, Oxford, England, 1983.

22. Cheriton, David R., "The V Kernel: A Software Base for Distributed Systems," *IEEE Software*, April 1984.

23. Cheriton, David R., and Timothy P. Mann, "A Decentralized Naming Facility," Stanford University, Computer Science Department, February 1, 1986.

24. Cheriton, David R., and Willy Zwaenepoel. "The Distributed V Kernel and its Performance for Diskless Workstations," *Proceedings of the Ninth ACM Symposium on Operating System Principles, ACM, October, 1983.*

25. Davis, C. T., "Data Processing Spheres of Control," *IBM System Journal* (1978) vol. 17, no. 2, pp. 179–198.

26. Druffel, Larry E., "Software Technology for Adaptable, Reliable Systems (STARS) Program Strategy," *ACM SIGSOFT Software Engineering Notes* (April 1983) vol.

8, no. 2, pp. 56-108.

27. Foudriat, E. C., W. J. Berman, R. W. Will and W. L. Bynum. "An Operating System for Future Aerospace Vehicle Computer Systems (preliminary)," Langley Research Center, NASA, Norfolk, Virginia, April 1984, p. 45.
28. Foudriat, E. C., "An Architecture for Embedded Network Computer Systems," Preliminary Draft, Internal NASA Report, Langley Research Center, NASA Norfolk, Virginia, 1985.
29. Foudriat, E. C., W. J. Berman, R. W. Will, "Design Considerations for Time-Critical Distributed Computer Systems," Langley Research Center, NASA, Norfolk, Virginia, July 1984, p. 45.
30. Grass, J. E., *Mediators*, Thesis, Tech Report, UIUCDCS,1986.
31. Grass, J. E. and R. H. Campbell, *Mediators: A Synchronization Mechanism,* **Proc. of Sixth International Distributed Computing Systems,** IEEE, Cambridge, Mass., May 19-23, 1986, pp. 468-477.
32. Grunwald, D. C., "An Implementation of Path Pascal," *MS Thesis,* Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1985.
33. Jalote, Pankaj and Roy H. Campbell, "Fault Tolerance Using Communicating Sequential Processes," In: *Digest of Papers, Fourteenth International Conference on Fault Tolerant Computing, IEEE FTCS14* (June 1984) pp. 347-352.
34. Jalote P. and R. H. Campbell, "Atomic Actions in Concurrent Systems," Proceedings of the *5th International Conference on Distributed Computing Systems,* Denver, May 1985.
35. Jalote P. and R. H. Campbell, *Atomic Actions for Fault-Tolerance using CSP,* **IEEE Transactions on Software Engineering, Special Issue on Software Reliability - Part II,** Vol. SE-12, No. 1., January 1986.
36. Jalote P., "Atomic Actions in Concurrent Systems," *Ph.D. Thesis,* Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1985.
37. Kim, K. H., "Approaches to Mechanization of the Conversation Scheme Based on Monitors," *IEEE Transactions on Software Engineering,* vol SE-8, no. 3, May 1982, pp. 189-197.
38. Kolstad, Robert Bruce. "Distributed Path Pascal: A Language for Programming Coupled Systems," *Ph.D. Thesis,* Department of Computer Science Technical Report #1136, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1983.
39. LeBlanc, Thomas J. "Programming language support for Real-Time distributed systems," *Proceedings, International Conference on Data Engineering* (April 1984) pp. 371-376.
40. Leinbaugh, Dennis W. "High Level Description and Synthesis of Resource Schedulers", Submitted to ACM '82: Ohio State University, Columbus, Ohio, 1982.

41. Liestman, A. and R. H. Campbell, *A Fault Tolerant Scheduling Problem,* **Digest of Papers FTCS–13: Thirteenth Annual International Symposium on Fault–Tolerant Computing,** Milano Italy, June 1983.

42. Liestman, A. and R. H. Campbell, *A Fault Tolerant Scheduling Problem,* **IEEE Transactions on Software Engineering,** October 1986.

43. Liskov, Barbara H. and Robert Scheifler. "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Transactions on Programming Languages and Systems,* vol. 5, no. 3, July 1983, pp. 381–404.

44. Lui, M. T., and R. C. Lian, "Cells: An Approach to Design of a Fault–Tolerant Network Operating System," 3rd Symposium on Reliability in Distributed Software and Data Base Systems, IEEE Comp. Soc. Press, Oct 17–19, 1983, pp. 163–172.

45. McKendry, M. S., "Ordering Actions for Visibility," *IEEE Transactions on Software Engineering,* vol 11, no. 6, June 1985, pp.509–519.

46. McKendry, M. S., and R. H. Campbell, *A Mechanism for Implementing Language Support in High–Level Languages,* **Transactions on Software Engineering**, Vol. SE–10, No. 3, May 1984, pp.227–236.

47. Merlin, P. M. and Brian Randell. "State Restoration in Distributed Systems," In: *Digest of Papers FTCS–8: Eighth Annual International Symposium on Fault–Tolerant Computing.*, Toulouse, 1978, pp. 129–134.

48. Mickunas, M. D., Pankaj Jalote and Roy H. Campbell. "The Delay/Re–Read protocol for Concurrency Control," In: *Proceedings, First International Conference on Data Engineering.* IEEE, Los Angles, California, 1984.

49. Moss, J. E. B., "Nested Transactions: An Approach to Reliable Distributed Computing," *Tech. Report,* MIT/LCS/TR–260, MIT, Cambridge, Mass., 1981.

50. Rashid, Richard F., "Accent: A Communication Oriented Network Operating System Kernel," *ACM SIGOPS,* Vol. 15, No. 5, 1981, pp. 64–75.

51. Ritchie, D. M. and K. Thompson, "The UNIX Time–Sharing System," *Comm. of ACM,* Vol. 17, No. 7, July 1974, pp. 365–375.

52. Schmidt, George Joseph. "The Recoverable Object as a Means of Software Fault Tolerance," *MS Thesis,* Department of Computer Science, University of Illinois at Urbana–Champaign, Urbana, Illinois, 1983.

53. Shrivastava, S. K. and F. Panzieri. "The Design of a Reliable Remote Procedure Call Mechanism," *IEEE Transactions on Computers* (July 1982) vol. C–31, no. 7.

54. Spector, Alfred. "Performing Remote Operations Efficiently on a Local Computer Network," *Communications of the ACM* (April 1982) vol. 25, no. 4.

55. Wei, A. Y., K. Hiraishi, R. Cheng, R. H. Campbell, *Application of the Fault–Tolerant Deadline Mechanism to a Satellite On–Board Computer System,* **Digest of Papers FTCS–10: Tenth International Symposium on Fault–Tolerant Computing,** Kyoto Japan, October 1980.

56. Wittie, L. D., "Communication Structures for Large Networks of Microcomputers," *IEEE Trans. on Computers*, Vol. C–30, No. 4, April 1981, pp.264–273.

57. Wittie, L. D. and A. Van Tilborg, "MICROS – A Distributed Operating System for MICRONET – A Reconfigurable Network Computer," *Tutorial, Microcomputer Networks*, ed. H. A. Freeman and K. J. Thurber, IEEE press, 1981, pp. 138–147.

# EOS Bibliography

Roy H. Campbell

Department of Computer Science

University of Illinois at Urbana–Champaign

Urbana, Illinois

*Project Eos Bibliography*


1. Campbell, R. H., K. Horton, and G. G. Belford, *Simulations of a Fault–Tolerant Deadline Mechanism,* **Digest of Papers FTCS–9: Ninth Annual International Symposium on Fault–Tolerant Computing**, Madison WI, June, 1979, 95–102.
2. Campbell, R. H. and R. B. Kolstad, *Path Expressions in Pascal,* **Proceedings of the Fourth International Conference on Software Engineering**, Munich, September 17–19, 1979, 212–219.
3. Campbell, R. H. and R. B. Kolstad, *Practical Applications of Path Expressions to Systems Programming,* **ACM79**, Detroit, 1979, 81–87.
4. Campbell, R. H. and R. B. Kolstad, *An Overview of Path Pascal's Design,* **Sigplan Notices**, Vol. 15, No. 9, pp. 13–14, September, 1980.
5. Kolstad, R. B. and R. H. Campbell, *Path Pascal User Manual,* **Sigplan Notices**, Vol. 15, No. 9, pp. 15–24, September, 1980.
6. Kolstad, R. B. and R. H. Campbell, *Directions for User Defined Communication for Distributed Software,* **Proceedings of The International Conference on Parallel Processing**, IEEE 80CH1569–3, pp. 188–189, Boyne MI, August 26–29, 1980.
7. Wei, A. Y., K. Hiraishi, R. Cheng, R. H. Campbell, *Application of the Fault–Tolerant Deadline Mechanism to a Satellite On–Board Computer System,* **Digest of Papers FTCS–10: Tenth International Symposium on Fault–Tolerant Computing**, Kyoto Japan, October 1980.
8. Kolstad, Robert Bruce. "Distributed Path Pascal: A Language for Programming Coupled Systems," *Ph.D. Thesis,* Department of Computer Science Technical Report #1136, University of Illinois at Urbana–Champaign, Urbana, Illinois, 1983.
9. Liestman, A. and R. H. Campbell, *A Fault Tolerant Scheduling Problem,* **Digest of Papers FTCS–13: Thirteenth Annual International Symposium on Fault–Tolerant Computing**, Milano Italy, June 1983.
10. Schmidt, George Joseph. "The Recoverable Object as a Means of Software Fault Tolerance," *MS Thesis,* Department of Computer Science, University of Illinois at Urbana–Champaign, Urbana, Illinois, 1983.
11. Campbell, R. H., *Distributed Path Pascal,* **In Distributed Computing Systems**, (Editor Y. Paker and J.–P. Verjus), Academic Press, 1983, pp. 191–224.
12. Campbell, R. H. and T. Anderson, *Practical Fault Tolerant Software for Asynchronous Systems,* **SAFECOMP 83, Third International IFAC Workshop on Achieving Safe Real–time Computer Systems**, Pergamon Press, Oxford, England, 1983.
13. Jalote, Pankaj and Roy H. Campbell, "Fault Tolerance Using Communicating Sequential Processes," In: *Digest of Papers, Fourteenth International Conference on Fault Tolerant Computing, IEEE FTCS14* (June 1984) pp. 347–352.

14. Campbell, Roy H., Jeff Donnelly, Raymond B. Essick, Judith Grass, Dirk Grunwald, Pankaj Jalote and David A. McNabb. "The Embedded Operating System Project," 1984 Mid–Year Report, NASA GRANT NSG 1471, Software Systems Research Group, Department of Computer Science, University of Illinois at Urbana–Champaign, Urbana, Illinois, 1984.

15. McKendry, M. S., and R. H. Campbell, *A Mechanism for Implementing Language Support in High–Level Languages,* **Transactions on Software Engineering**, Vol. SE–10, No. 3, May 1984, pp.227–236.

16. Mickunas, M. D., Pankaj Jalote and Roy H. Campbell. "The Delay/Re–Read protocol for Concurrency Control," In: *Proceedings, First International Conference on Data Engineering.* IEEE, Los Angles, California, 1984.

17. Jalote P. and R. H. Campbell, "Atomic Actions in Concurrent Systems," Proceedings of the *5th International Conference on Distributed Computing Systems*, Denver, May 1985.

18. Grunwald, D. C., "An Implementation of Path Pascal," *MS Thesis,* Department of Computer Science, University of Illinois at Urbana–Champaign, Urbana, Illinois, 1985.

19. Jalote P., "Atomic Actions in Concurrent Systems," *Ph.D. Thesis,* Department of Computer Science, University of Illinois at Urbana–Champaign, Urbana, Illinois, 1985.

20. Campbell, Roy H., Raymond B. Essick, Judith Grass, Dirk Grunwald, Pankaj Jalote, Kevin Kenny and David A. McNabb. "The Embedded Operating System Project," 1985 Mid–Year Report, NASA Grant NSG 1471, Software Systems Research Group, Department of Computer Science, University of Illinois at Urbana–Champaign, Urbana, Illinois, 1985.

21. Jalote P. and R. H. Campbell, *Atomic Actions for Fault–Tolerance using CSP,* **IEEE Transactions on Software Engineering, Special Issue on Software Reliability – Part II,** Vol. SE–12, No. 1., January 1986.

22. Grass, J. E., *Mediators*, Thesis, Tech Report, UIUCDCS,1986.

23. Grass, J. E. and R. H. Campbell, *Mediators: A Synchronization Mechanism,* **Proc. of Sixth International Distributed Computing Systems,** IEEE, Cambridge, Mass., May 19–23, 1986, pp. 468–477.

24. Campbell R. H. and B. Randell, **Error Recovery in Asynchronous Systems, IEEE Transactions on Software Engineering, August,** 1986.

25. Liestman, A. and R. H. Campbell, *A Fault Tolerant Scheduling Problem,* **IEEE Transactions on Software Engineering,** November 1986.

# Mediators: A Synchronization Mechanism

J.E. Grass

R. H. Campbell

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois

Presented at the 6th International Conference
on
Distributed Computing Systems
May 1986

# MEDIATORS: A SYNCHRONIZATION MECHANISM

J. E. Grass and R. H. Campbell

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois

**Abstract**

This paper describes a construct called a *mediator*. Mediators support synchronization and scheduling for systems programming within distributed systems. Mediators are based on a resource view of systems, and fit within a programming methodology that emphasizes resource modularity, synchronization modularity and encapsulated concurrency. The paper examines other existing synchronization mechanisms in the light of modular programming requirements. Subsequently, a sample syntax and semantics for mediators is presented with many examples.

The mediator includes many interesting features. These include: an adaptation of guarded commands; *keys* that allow requests to be examined and manipulated before they receive service; parallel guard execution; coupled and uncoupled modes of service execution.

Finally, the paper discusses a few aspects of implementation.

## 1. Introduction

This paper introduces the *mediator* construct for implementing synchronization and scheduling in distributed systems. This language construct supports systems programming applications that require complex and flexible synchronization and scheduling schemes. The research was prompted by the recognition that many of the existing language constructs either overly constrain concurrency, make expression of some kinds of synchronization and scheduling difficult, or due to formal language design considerations fail to provide practical support for real programmers. The discussion of design goals that follows indicates examples of each of these failings.

### 1.1. The Problem

The development of the *mediator* was motivated by the lack of synchronization and scheduling tools to adequately support the development of distributed systems, such as those embedded in space craft. Such tools must meet a number of requirements, including support for modular and structured system design, flexibility, expressiveness, clarity and ease of use.

Modular design is a powerful aid to structuring software development which affects all phases of the software life cycle from specification, through development, testing and validation to maintenance. These three aspects of modularity must be considered: resource modularity, encapsulation of concurrency and synchronization modularity.

Resource modularity is a basic concern in both sequential and concurrent program design. The development of abstract data types [1] and object–oriented programming [2] are an expression of this concern [3]. The encapsulation of data and controlled access through carefully defined operations provide the user with a higher-level, abstract view of a data resource[4]. At the same time, the data is protected from invalid accesses. The module also creates a locality of reference, placing the data and operation definition in one place rather than scattered throughout the code.

Early synchronization tools, including busy–waits, semaphores [5], and conditional critical regions [6–8], did not create a locality of reference, and so made the structuring of synchronization difficult. Most recent proposals have recognized this problem, and have taken some version of the abstract data type as a base. In some cases the module is a passive and takes no action until called on by an active process (e. g. monitors [9] ). Passive synchronization modules are the rule in constructs based on shared data. Usually constructs based on message passing use an active module. Ada [10], Distributed Processes (DP) [11], Synchronizing Resources (SR) [12], and Argus [13] belong in this category.

CSP [14] also uses a message passing model, but it is not strictly based on an abstract data type model. In CSP, individual processes encapsulate data. Other processes may access the encapsulated data only by an exchange of messages. The process owning the data resource defines all the operations on the data and localizes data access. Synchronization is not as well localized, because the synchronization depends on the "matching" of input and output commands distributed among many processes.

Although there are many synchronization constructs that support resource modularity, relatively few of them permit real concurrency within the encapsulated module.

This work is part of the EOS project and was supported in part by NASA grant NSG–1471.



CH2293-9/86/0000/0468$01.00 © 1986 IEEE

For instance, monitors allow at most one process to be active at a time. In order to allow multiple processes to access a resource simultaneously (as for reader processes in the well-known readers and writers problem [15] ), a monitor is used only to implement a pre-read/ post-read and pre-write/ post-write protocol, which is called before and after a call to an external *read* or *write* routine [9]. There is no assurance that the protocol will be followed. Deadlock or data corruption may result if it is not. The lack of encapsulated concurrency also makes it difficult to nest modules or to otherwise structure concurrency. Structured concurrency is needed to develop atomic action and fault-tolerant systems [13,16–18]. Concurrent Pascal (which is monitor based) [19], DP [11], Ada [10] and CSP [14] all fail to encapsulate concurrency. Argus [13,20] provides encapsulated concurrency, but with severe restrictions to ensure recoverability. SR [12], Path Pascal (PP) [21] Distributed Path Pascal (DPP) [22], serializers [23], and MCP [24] do allow specification of encapsulated concurrency.

Synchronization modularity refers to the ability to specify synchronization and scheduling constraints separate from the specification of the resource data abstraction. This additional structuring device aids in system development, but also benefits the validation of design and code. Modular synchronization may also make it possible to develop libraries of synchronizers and schedulers. The isolation of timing aspects contributes to real-time programming as well.

Few constructs provide synchronization modularity. Among those are Path Pascal (PP and DPP) [21], sentinel processes [25], and serializers [23]. Serializers are implemented in a LISP environment. Sentinel processes appear to be the imperative language analog. Both combine built-in counters with queueing primitives to allow modular specification of synchronization. These constructs appear to be well suited to FIFO scheduling problems and variants of the reader/writer problem, but are less flexible than desired [26]. Path Pascal encapsulates most synchronization specifications in a path expression. This often provides a high degree of synchronization modularity. The synchronization modularity is lost when conditional synchronization or scheduling is specified. These must be programmed using nested objects. This results in loss of modularity as well as inefficiency due to the implicit scheduling applied at each level of nesting. In order to maintain synchronization modularity, synchronization data must be encapsulated. In addition, there must be support for conditional synchronization and scheduling.

For practical embedded distributed systems, it is important not to overly constrain the system implementor in terms of possible synchronization and scheduling. Synchronization schemes that enforce atomic recoverable transactions (such as Argus and Clouds [27,28]) are overly conservative in what can be specified. These systems use lock-based synchronization schemes to ensure serializability of actions in order to allow recoverability if an action should fail. Much of this is not directly in control of the action implementor. In this sense, these schemes are overly restrictive.

Other schemes allow more flexibility in what can be specified, but make the expression of some kinds of constraints difficult. As we noted above, Sentinel Processes make scheduling problems quite easy to specify, but specification of operation sequences is complicated [25]. In Path Pascal it is easy to specify sequences of operations, but implementing scheduling or conditional synchronization is complicated. It should be possible to express constraints in terms of resource history, resource and synchronization state and information about pending requests.

The configuration of concurrent systems raises other questions about flexibility. Many proposed language constructs for writing distributed systems rely on static systems. In DP and Concurrent Pascal [11,19] processes and modules are instantiated at system creation and never terminate. This is not reasonable for real systems that sometimes require on-the-fly reconfiguration to add newly developed services; nor does this adequately allow processes to abnormally terminate due to an error. DP and Concurrent Pascal do not support dynamic allocation and reallocation of resources.

Other constructs allow processes and objects to come and go, but are inflexible in other ways. Frequently communication paths are static. CSP is an extreme case of this [14] in which the sender and the receiver of a message need to know each other's name. This feature of CSP makes it difficult to write libraries of services [14]. CSP was meant to be an exercise in programming using input, output and concurrency primitives rather than a complete language proposal [14]. Some CSP successors, such as OCCAM [29], have attacked this problem by introducing ports. SR [30] has a similar communication problem. Server processes and clients are tied in a one-to-one relationship that is explicit and rigid.

Most synchronization proposals allow servers to honor requests from anonymous clients. This is a flexible arrangement, but occasionally there are cases in which the client's identity must be known. Some language constructs provide this information (PLITS [31] ), but more often it is left up to the implementor. The *mediator* proposal supports dynamic creation and termination of mediators and flexible communication paths. It also provides a means of identifying clients.

### 1.2. A Proposal

The *mediated object* combines several proposals in an attempt to provide a solution to the problems that are outlined above. The mediated object paradigm is based on object-oriented language design for operating systems applications. In this model, resources are encapsulated and access to them is allowed only through exported operations. The synchronization schemes used in DP [11], Monitors [9], SR[12] and Ada [10] all are examples of languages using this paradigm. The mediated object

encapsulates data and allows access to that data through a well-defined interface. Client processes request a service from an exported list of service names, and the mediator determines how the service will be provided. Synchronization and scheduling constraints are specified by the mediator body, and isolated from the definition of data and operations.

The main features of a mediated object are given below.

1) Initialization and termination blocks are included both for the data resource and for the mediator.

2) The essential control structure within the mediator is an adaptation of Dijkstra's guarded commands [32]. Our adaptation uses *delay semantics* [11] rather than Dijkstra's *abort* semantics.

3) Requests are associated with unique *keys* that allow the mediator to manipulate requests and implement scheduling.

4) Guards may contain *status tests* to inquire about pending requests, and boolean tests which may refer to data contained in pending requests [14,31].

5) The mediator controls execution of client requests by commands allowing coupled and uncoupled client process execution [33]. There is an explicit command to return results to a client.

6) *Parallel guards* are used to multi-program the mediator. Mediator execution is guaranteed mutually exclusive between guard evaluations.

7) Mediators map the name of a service requested by the client onto that of an appropriate operation. Clients do not call on services directly.

The proposal presented here is preliminary. A formal definition of mediators using temporal logic is in preparation. Some features of the syntax and semantics may change as the formal description is developed, and as implementation issues become more central. Section two of this paper explains the mediator in greater detail, presenting examples. Section three examines implementation aspects. Many of the individual components of the mediator have been implemented in other languages. The main difficulty is combining these in an efficient manner.

## 2. Concepts and Notations

*Mediators* and *mediated objects* are built out of a small number of concepts combined to provide a means of implementing distributed systems resources. The *mediated object* is one component of a larger language. This paper does not present a complete language. The "host" language is assumed to be similar to Pascal.

### 2.1. The Mediated Object

The mediated object includes the definition of encapsulated data and operations defined on that data as well as the specification of the mediator itself. The following is a schema of a mediated object.

```
name = object
   interface declaration

   resource variables
   resource operations

   mediator
      mediator variables
      mediator routines
        initialisation block
        mediator body
        termination block
   end mediator
end object
```

A mediated object is made up of three parts: 1) the interface, 2) the encapsulated resource and 3) the mediator. The resource constants, types and variables defined within the object are shared by the resource routines. The mediator maps requests for services listed in the interface onto appropriate operations and synchronizes access. The mediator may contain its own data and routines not accessible to any external caller. Mediator data usually consists of flags and counters, although it may also include queue structures for scheduling.

The mediated object is a type, and a user may create several instantiations of a given object. The mediator initiation code is executed when an object is instantiated. The termination code executes when the body of the mediator terminates.

Figure 1 presents a complete mediated object. In other examples, only the mediator will be presented. Figure 1 contains many notations that have not yet been explained. It illustrates the declaration of an interface, object data (*RW_data*), resource routines (*read* and *write*), and local mediator data (*reader_count*). Object parameters are passed by value and by value–result. Reference parameters seriously compromise data encapsulation and are impractical for current distributed implementations.

Clients request a mediator service which is named in the interface by including the name of the service as a parameter to a call on the object. Once a client process has requested a service, the client is blocked until the mediator returns the results of the completed service. The actual execution of a requested service may be delayed by the mediator. The semantics of a call on a mediator is the same, whether the mediator is installed at a remote location or locally.

### 2.2. Basic Mediator Statements

The mediator is composed of several kinds of basic statements and a specialized control structure. The simple statements that can be used within the mediator include: assignments, local mediator routine calls, and the commands skip, exec, spawn and release. Exec, spawn and release are statements to initiate services for clients and to return the results of services. These have a *key* variable parameter that uniquely identifies the client for which the action was taken. *Keys* are explained in detail below. The second parameter of an exec or spawn

```
reader_writer = object
  interface
    job : export part
      pid : key client_process_id;
      case service : (read, write) of
        read : (readprm: var some_type);
        write : (writeprm :        some_type);
    end export part;

  var RW_data: some-type;

  procedure read (readprm: var some_type);
      begin readprm := RW_data end procedure;

  procedure write (writeprm : some_type);
      begin RW_data := writeprm end procedure;

  mediator
      var
        reader_count : integer;
        i, j         : client_process_id;
      init reader_count := 0 end init
      body
        any i in key:
            cycle
              req(i); job(i).service = write ->
                  cycle
                    reader_count = 0 ->
                        exec(i, write (job(i).writeprm));
                        release(i);
                  until true
              □
              req(i); job(i).service = read ->
                  reader_count := reader_count + 1;
                  spawn(i, read (job(i). readprm));
            until false
        //
        any i in key:
            cycle
              term(i); job(i).service = read ->
                  reader_count := reader_count - 1;
                  release(i);
            until false
      end body
  end mediator
end object
```

Figure 1. Reader_Writer Object.

statement is a resource operation call. Exec permits *coupled* execution of a resource operation (on behalf of a client identified by the *key*). The mediator initiates a process to execute the operation, and then blocks until the operation has terminated. For example, in the *reader_writer* object above, the statement *exec(i, write (job(i). writeprm));* initiates a *write* operation for client *i*. The mediator blocks until the operation has completed. On the other hand, spawn initiates an operation and allows *uncoupled* execution. The mediator does not wait for the operation to terminate, and continues executing mediator code. In the *reader_writer* object, the statement *spawn(i, read (job(i). readprm);* initiates a *read* operation for client *i*.

The release command returns the results of an operation to the client and removes the request from the mediator. This may be invoked only after an exec has been completed, or a status test (term, see Section 2.3) reveals that a spawned request has terminated. *Reader_writer* (figure 1) contains examples of release both after coupled and uncoupled service. The separate termination test allows synchronization data to be maintained as services complete. Release also makes it possible to delay and synchronize termination and the return of results. This can be used to implement a conversation scheme [34], atomic actions[35], or other forms of fault-tolerance.

### 2.3. Guarded Commands

Sequences of actions within the mediator body are specified by the control structures presented here, and by parallel guarded commands, which are presented in section 2.4. The basic mediator control structure is a guarded command of this form:

```
any identifier in key:
  cycle
     guard  ->  statement_list;
     □
        ...
     □
        guard  ->  statement_list
  until exit_condition;
```

The prefix any ... key: is optional.

The mediator guarded command has many similarities to Hoare's CSP guarded commands [14], which in turn can be credited to Dijkstra [32]. The chosen keywords and semantics are closer to the guarded regions of Brinch Hansen's DP [11]. The concept of *key* is related to Hoare's guard command range [36], and to message keys in PLITS [31,37]. The similarities and differences will be discussed below.

A guarded command is a control statement in which different statement lists are chosen for execution based on the truth value of the associated guards. Because the evaluation of guards is central to this construct, they will be explained first. The guarded command will be described after. The application of keys to guarded commands will be presented last.

Guards are made up of a *status test* and boolean equations. Mediator guard evaluation always results in either a *true* or a *false* value. The special guard otherwise is *true* only when all the other guards in the guard command are *false*.

*Status tests* allow inquiries about pending requests for mediator service. These are tests for requests to initiate an operation (req) or to return results after the operation has completed (term). For the guard *req( i )* to be true, the list of unserved requests must contain a request from client *i*. Once the guard has been *fired* (it's associated statement list chosen to execute), *req( i )* cannot become true again until the service has been completed and the results returned (by *release( i )*). The guard *term( i )* is similar, becoming true when the execution of an operation for client *i* terminates.

A boolean guard paired with a status test may examine the value of a client's request parameters. Each

client's request is represented within the mediator by a job descriptor defined by the interface declaration. The descriptor is a variant record containing fields for a key variable, the name of the service requested and the parameters for that service. The service field serves as a tag for variant parameter fields. The descriptor is accessed using the key by indexing on the variable *job*, as in these examples. The job descriptor for the *reader_writer* object is defined by the interface section in figure 1. In the *reader_writer* object, *job(i).service* references the service tag field. Boolean guards may also test the value of the mediator's local variables. Boolean guards paired with status tests are not evaluated if the status test is false.

In the following explanation of a guarded command, the execution of the guard is considered in isolation, without considering possible interleaving with other parallel guarded commands. The presence of parallel guards introduces delays, but does not affect the semantics of the guarded command.

Mediator guarded commands are closely related to Brinch Hansen's guarded regions [11]. The mediator process must wait until some guard condition is true, and then execute the associated statement list. A statement list associated with a *true* guard is said to be *enabled*. A guard whose associated statement list has been chosen and started execution is said to have been *fired*.

When the statement list of a fired guard has finished executing, the exit condition in the final until line of the guarded command is tested. If the condition is true, the guarded command terminates, otherwise its guards are reevaluated.

Nondeterminism is a possibility when more than one guard is enabled. In this case, one guard will be chosen to fire. A mediator implementation must ensure at least weak fairness to avoid starvation problems. The mediator cannot delay if there are enabled guards.

The *delay* semantics of this guard command differs from Dijkstra's original definition and Hoare's adaptation [14,32]. Hoare and Dijkstra's constructs abort the guarded command when no guard is true. This creates a framework that is convenient for formal verification, but results in servers that do not facilitate waiting. Waiting is usually implemented by explicitly programming a busy loop. Because waiting is fundamental to providing services, we prefer to wait implicitly.

Brinch Hansen implements both *delay semantics* in guarded regions and *abort semantics* for guarded commands. The mediator proposal includes only *delay semantics*, because the inclusion of an otherwise guard and exit conditions make the abort semantics redundant. The otherwise guard has other applications for implementing background actions and is a useful shorthand for the negation of all other guards.

Mutual exclusion within a mediator depends both on the use of the exec statement and the careful choice of preconditions defined in a guard statement. The exec statement initiates a service process and blocks the mediator, but it does not check for other initiated processes. In the Reader–writer example (figure 1), mutual exclusion for the *write* operation is ensured by the guard "cycle *reader_count = 0 ->*" and by the action of exec. The guard will not permit a *write* to begin until all executing *read* operations are terminated. The exec statement blocks the mediator as the *write* is serviced to prevent other operations from becoming active.

*Keys* are used to identify the client to the mediator, to access job descriptors for guard evaluation and scheduling purposes and to tie clients to specific resources, as in allocator objects. The *key* concept was suggested by Hoare's CSP process range labels [14,36], but their use in mediators is considerably different. Hoare applies ranges to processes to create a finite number of explicitly and contiguously indexed processes. This application of ranges is not included in mediators. Hoare also applies ranges to guarded commands to substitute values within a given range for a bound variable in the guard statements. The following example is from [14]:

"(*i*:1..*n*)G -> CL stands for
G1 -> CL1 □ G2 -> CL2 □ ... □ Gn-> CLn."

In effect, the guard is expanded by creating a guard and statement list for every value of *i*. The application of ranges in Hoare's guarded commands is quite general.

In the mediator proposal, keys serve only to identify client processes. Like Hoare's ranges, a key statement (any ...) defines a key variable which will be bound within the guard command it modifies. Consider the guarded command shown in figure 2. It is executed as if it were written as shown in figure 3. In this example the value of the key identifier is in the range *1..n* and defined as the interface field *job.rangeprm*. Usually a process identifier (the *pid* descriptor field) will be used as the key. The designer of a mediator does not need to know explicitly what process identifier values are being used, just that they are unique. Although, in an abstract sense, a potentially infinite key variable range implies an infinitely expanded guard, there is no need to implement them that way. Keys are always associated with status tests. Only guards corresponding to clients with requests can evaluate to *true*, so only such guards need to be evaluated. This significantly limits the number of guards evaluated. Evaluation can be restricted further when fairness is taken into consideration.

Key variables are tied to job descriptors defined by the interface. The most useful key reference is to the client process identifier. The mediator designer may designate another descriptor field as a key, as in figures 2 and 4. In any case, the chosen key field must be unique for each pending request.

The mediator in figure 4 implements synchronization for the dining philosophers problem. The client process executes the statement *diner (rangeprm, eat);* to request the mediator's *eat* service. This solution is one of many possible solutions using mediators.

ORIGINAL PAGE IS
OF POOR QUALITY

ORIGINAL PAGE IS
OF POOR QUALITY

```
any i in key:
        cycle
                req(i); job(i).service = A ->
                        exec(i,A);
                        release(i);
                □
                req(i); job(i).service = B ->
                        x := x + 1;
                        exec(i,B);
                        release(i)
        until false;
```

Figure 2. A Guarded Command.

```
cycle
        req(1); job(1).service = A ->
                exec(1,A);
                release(1);
        □
        req(1); job(1).service = B ->
                x := x + 1;
                exec(1,B);
                release(1);
        □
        req(2); job(2).service = A ->
                exec(2,A);
        ...

        □
        req(10); job(10). service = B ->
                x := x + 1;
                exec(10.B);
                release(10)
until false;
```

Figure 3. The Guarded Command Expanded.

## 2.4. Parallel Guarded Commands

The following schematic shows the syntax of the parallel guarded command, a mechanism that allows the interleaving of different mediator actions.

```
body
        guarded_command
        //
        ...
        //
        guarded_command
end body
```

Parallel guarded commands are proposed to allow different sets of guards to be evaluated at different times during mediator execution. It allows the mediator to "shuffle" together the evaluation of several guarded commands. The choice of the notation // to separate parallel guarded commands is deliberate. A mediator containing parallel guarded commands uses a multiprogrammed thread of control, one thread of control for each guarded command. Only one thread of control is active at a time. The active control block can change only when guards are evaluated. This creates mutually exclusive execution of the statement lists between guard evaluations. The mediator body terminates if all of the parallel guard blocks terminate.

Consider the simplified example in figure 5. (Labels have been included to make discussion easier). In figure 5, *A, B, C, D* are guards. *SA, SB, SC, SD* are statement lists. The control vector of this mediator has two elements. The notation "*<label1, label2, ... , labeln >*" is a control vector in which n threads of control are at the locations label1 through labeln. This notation is adapted from the expression of execution state in Manna and Pnueli's temporal logic scheme[38]. In figure 5, the initial control vector is: <l1, m1>. When guard evaluation occurs in the initial state, the guards *A* and *C* are evaluated. As for isolated guard commands, the associated statement list of some true guard will be executed. If the guard *A* from the cycle l1 is fired, the statement list starting at l2 will begin execution. It will continue executing without interruption until the new guard command at l3 is encountered (assuming SA contains no

```
interface
  job: export part
    myfork : key range;
    case service: (eat) of
          eat: ()
  end export part;

mediator
  type range = 0 .. n-1
  var
          fork : array [range] of (free, inuse);
          i, j   : range;
  init
       for j := 0 to n-1;  fork[j] := free;
  end init

  body
       any i in job.rangeprm :
          cycle
               req(i); job(i).service = eat and fork[i] = free
                       and fork[(i+1) mod n] = free ->
                  fork[i] := inuse;
                  fork[(i+1) mod n] := inuse;
                  spawn(i, eat(i));
               □
               term(i); job(i).service = eat ->
                  fork[i] := free;
                  fork[(i+1) mod n] := free;
                  release(i);
          end cycle
  end body
end mediator
```

Figure 4. Dining Philosophers.

```
body
  l1: cycle
        A ->   l2: SA;
               l3: cycle  B ->   l4: SB  until true;
    until false
      //
  m1: cycle
        C ->   m2: SC;
               m3: cycle  D ->   m4: SD  until true;
    until false
end body
```

Figure 5. Simplified Parallel Guarded Command.

guard commands). At this point the control vector is <13, m1>, and the new guard evaluation includes the guards *B* and *C*. Considering all possible combinations, the set of guards evaluated at any one time may be: [A, C], [A, D], [B, C] or [B, D].

The statement lists following guards may contain exec, spawn and release statements without altering the flow of control discussed above. In every case, control passes to the following statement. In the case of an exec statement, this is delayed until the resource operation it has initiated terminates. This delay temporarily blocks further mediator activity, but does not alter the flow of control.

The parallel guard notation is an easy and concise way of specifying changing sets of enabling conditions. It is possible to rewrite a parallel guard as one large simple guard command by using a distribution algorithm. The resulting guard command is considerably more bulky and actually less clear.

The introduction of a control vector within the mediator does not create the same complications for reasoning about programs that are usually associated with parallel processes. The control flow in mediators is very restricted, giving statement lists that will be executed in mutual exclusion. This fact, combined with the small size of mediators and the explicit statement of preconditions in the guards makes it quite easy to reason about the behavior of parallel guards.

The reader/writer mediator demonstrates one application of the parallel guard. In that example, firing the guard *req(i); job(i).service = write* executes the associated statement, which is a *cycle* statement. As long as its guard *reader_count = 0* is false, the guard cannot fire. No new *write* or *read* operations will be initiated, but the second parallel guard will allow *read* operations to finish up and leave the mediator. Parallel guarded commands coupled with nested guard commands gives a convenient way to block some actions while permitting others.

### 2.5. Some Additional Examples

The examples that follow demonstrate some applications of mediators. Only the mediator portion is included.

#### 2.5.1. Alarm clock

The alarm clock object (figure 6) delays a caller for a time period specified in the request's parameter *n*. Calls for the *wake* service cause a delay. Calls for the *tick* service advance the clock. The field *out_time* must be declared for the operation *wake* job descriptor within the mediator as a mediator local extension to the job descriptor.

#### 2.5.2. Shortest Job Next

The mediator in figure 7 implements a scheduler that chooses the job with the lowest estimated service time for the next execution. Requests are served in mutual exclusion. This framework is applicable to many scheduling problems.

```
init now := 0 end init
body
  any i in key:
      cycle
        req(i); job(i).service = wake ->
              -- start the service, but termination will be delayed
              job(i). out_time := now + job(i). n;
              spawn(i, wake);
      until false;
      //
  any i in key:
      cycle
        req(i); job(i).service = tick ->
              now := now + 1;
              exec(i, tick);
              release(i);
              flag := false;
              any j in key:
                cycle
                      term(j); job(j).service = wake
                            and job(j).out_time <= now ->
                        release(j);
                      □
                      otherwise -> flag := true
                            -- exit cycle
                until flag;
      until false;
end body
```

Figure 6. Alarm Clock.

```
body
  any i in key:
      cycle
        req(i); job(i).service = server ->
              enqueue (i, job(i).estimate);
      until false;
      //
  cycle
      queue_not_empty ->
        j := dequeue;
        spawn(j, server);                -- initiate service operation
        cycle
              term(j); job(j).service = server ->
                release(j);
        until true;
  until false;
end body
```

Figure 7. Shortest Job Next.

The first guard command simply calls a local operation to queue up job descriptors in order of their estimate parameter. The second guard command removes the head element of the job descriptor queue and starts that job's execution. The *spawn* and wait for termination allows the mediator to continue enqueueing new requests while a service operation is executing.

The key variable *j* in the second guard command is set by direct assignment rather than through a cycle modifier.

#### 2.5.3. An Allocator

The allocator in figure 8 gives a client process exclusive rights to a resource for a series of accesses. The

```
body
  any i in key:
      cycle
        req(i); job(i). service = allocate ->
              exec(i, allocate);
              release(i);
              flag := false;
              cycle
                 req(i); job(i).service = use ->
                       exec(i, use);
                       release(i);
                 □
                 req(i); job(i).service = free ->
                       exec(i, free);
                       release(i);
                       flag := true;
              until flag;
      until false;
end body
```

Figure 8. Allocator.

client must request an allocation, then may make repeated calls on the resource. Finally, the client must explicitly release the resource before it can become available to another client. This example uses the key binding made in the outer cycle to restrict use of the resource to one process in the inner cycle.

The mediator differs from the monitor solution [9] for this problem in a number of ways. Most importantly, the resource being allocated is encapsulated with the mediated object. The mediator protects the resource from unsynchronized accesses by faulty processes. The mediator also prevents the resource from being released by any process but the one the resource has been granted to. The monitor solution does not offer protection in either of these cases.

## 3. Implementation

Implementing mediators should not present significant problems, because many of the components of the construct have been implemented in other languages. The main problem will be fitting the components together in an efficient manner.

There are several possible implementations for the mediator call mechanism. For example remote procedure calls could be applied [39]. A remote procedure call can be implemented as an exchange of messages between the client and mediator. The client sends a request message containing the name of the operation requested, its process identifier and parameters. It then waits to receive a reply, which will arrive when the mediator has released the operation. The mediator receives a request and creates a job descriptor. This is placed in the list of pending requests, becoming available for status tests. The job descriptor is destroyed when the mediator releases a job and returns results to the client. In the perception of the client process, a remote procedure call appears to be no different than a simple local procedure call.

The exec and spawn statements require system support to initiate service for requests. This support may include creating a new system process and scheduling its execution on a free processor.

The special application of guards in mediators makes it possible to limit the number of guard reevaluations. After a guard evaluation, only certain events may change the value of the guards: the arrival of a new request, the termination of an active request or the execution of mediator statements after a guard has fired. If all guards have evaluated as false, there is no need to reevaluate the guards until either new requests arrive, or active requests terminate.

It is also possible to limit the number of guards considered during evaluation. The evaluation of guards containing status tests can be constrained in two ways. Status tests need only be evaluated for clients that are present in the mediators list of pending requests, since the value of any other status guard is automatically *false*. Application of fairness limits the evaluation of status tests for clients as well. These can be evaluated in the order of their arrival until an enabling guard is found.

The evaluation of pure boolean guards cannot be limited this way. Fortunately, these are likely to be few in number. These also present a fairness problem. It is easy to apply a fair ordering criteria for requests based on time of arrival, but such criteria can not be applied to simple boolean guards that may, without firing, become *true* and *false* repeatedly. Implementing weak fairness may require implementing event queues or counts so that these guards may be ordered.

The design of mediators is best suited to a system made up of distributed multiprocessor nodes, with one or several mediated objects installed at each node. Implementing mediators on such a system should be straightforward. Implementation of mediators on a uniprocessor is also possible using multiprogramming, but would probably be very inefficient. Mediators implemented on a distributed network of uniprocessors could work quite well. This could be accomplished by multiprogramming the mediated object on one node, or by allowing the mediator to exist on one node, and execute operations at remote nodes. The limiting factor would be the amount of object data that would need to be sent to the remote service nodes.

## 4. Conclusion

This paper has presented a preliminary proposal for a new language construct, the *mediator*, that may serve as a useful tool in programming distributed embedded systems. Mediators allow direct programming of synchronization and scheduling and are able to directly use both information about a pending request and the present synchronization state. This makes mediators a powerful construct for synchronization and scheduling applications. At the same time, the design of mediators supports structured design of concurrent programs.

Finally, mediators should not present significant implementation problems and are adaptable to a number of distributed architectures.

## References


1. Parnas, D. L. *A Technique for Software Module Specification with Examples.* CACM (May 1972) vol. 15, no. 5, pp. 330–336.
2. Birtwistle, G. M., O. J. Dahl, B. Myhrhaug and K. Nygaard. Simula Begin. Auerbach, Philadelphia, PA, 1973.
3. Dijkstra, E. W. A Discipline of Programming. Prentice-Hall, Englewood Cliffs, NJ, 1976.
4. Campbell, R. H. and A. N. Habermann. *The Specification of Process Synchronization by Path Expressions.* In: LNCS, vol. 16. Springer-Verlag, New York, NY, 1974, pp. 89–102.
5. Dijkstra, E. W. *Cooperating Sequential Processes.* In: Programming Languages, F. Genuys, ed. Academic Press, New York, NY, 1968.
6. Hoare, C. A. R. *Towards a Theory of Parallel Programming.* In: Operating Systems Techniques, C. A. R. Hoare and R. H. Perrott, ed. Academic Press, London, 1972, pp. 61–71.
7. Brinch Hansen, P. *Structured Multiprogramming.* CACM (July, 1972) vol. 15, no. 7, pp. 574–578.
8. ——. *Concurrent Programming Concepts.* ACM Computing Surveys (Dec. 1973) vol. 5, no. 4, pp. 223–245.
9. Hoare, C. A. R. *Monitors: An Operating System Structuring Concept.* CACM (Oct. 1974) vol. 17, no. 10, pp. 549–557.
10. Defense, U. S. Department of. *Programming Language Ada: Reference Manual.* In: Vol. 106 Lecture Notes in Computer Science. Springer-Verlag, New York, NY, 1981.
11. Brinch Hansen, Per. *Distributed Processes: A Concurrent Programming Concept.* CACM (Nov. 1978) vol. 21, no. 11, pp. 934–941.
12. Andrews, Gregory R. *Synchronizing Resources.* ACM TOPLAS (Oct. 1981) vol. 3, no. 4, pp. 405–430.
13. Liskov, Barbara and Robert Scheifler. *Guardians and Actions: Linguistic Support for Robust, Distributed Programs.* ACM Tran. Prog. Lang. and Syst. (July 1983) vol. 5, no. 3, pp. 381–404.
14. Hoare, C. A. R. *Communicating Sequential Processes.* CACM (Aug. 1978) vol. 21, no. 8, pp. 666–677.
15. Courtois, P. J., F. Heymans and D. L. Parnas. *Concurrent Control with Readers and Writers.* CACM (Oct. 1971) vol. 14, no. 10, pp. 667–668.
16. Campbell, Roy H. and Brian Randell. "Error Recovery in Asynchronous Systems", Tech. Report: Univ. of Illinois, Urbana-Champaign, Dept. Comp. Sci., Urbana, IL, 1984.
17. Jalote, Pankaj and Roy H. Campbell. *Recoverability of Actions and Atomicity.* IEEE TOSE (To appear in 1985).
18. Best, E. and B. Randell. *A Formal Model of Atomicity in Asynchronous Systems.* Acta Informatica (1981) vol. 16, pp. 93–129.
19. Brinch Hansen, P. *The Programming Language Concurrent Pascal.* IEEE TOSE (1975) vol. SE-1, pp. 199–206.
20. Weihl, William and Barbara Liskov. *Specification and Implementation of Resilient, Atomic Data Types.* SIGPLAN Notices (June 1983) vol. 18, no. 6, pp. 53–64.
21. Campbell, R. H. and R. B. Kolstad. *An Overview of PATH PASCAL's Design.* SIGPLAN Notices (Sept. 1980) vol. 15, no. 9, pp. 13–14.
22. Kolstad, Robert Bruce. "Distributed Path Pascal: A Language for Programming Coupled Systems", Ph. D. Thesis, Tech. Report: Dept. Comp. Sci., University of Illinois at Urbana-Champaign, UIUCDCS-R-83-1136, Urbana, IL, 1983.
23. Hewitt, C. E., and R. R. Atkinson. *Specifications and Proof Techniques for Serializers.* IEEE TOSE (1979) vol. SE-5, no. 1, pp. 10–23.
24. Bahsoun, H., C. Betorne and L. Feraud. *Une Expression de la Synchronisation et de l'Ordonnancement des Processus Concurrents par Variables Partagees.* In: Proc. 6th Int. Symp. on Programming: LNCS 167, M. Paul and B. Robinet, ed. Springer-Verlag, New York, NY, 1984, pp. 13–22.
25. Ramamritham, Krithivasan and Robert M. Keller. *Specifying and Proving Properties of Sentinel Processes.* In: Proc. 5th Int. IEEE/ACM Soft. Eng. Conference., 1981, pp. 374–381.
26. Bloom, Toby. *Evaluating Synchronization Mechanisms.* In: Proc. 7th Symposium on OS Principles (Pacific Grove, CA. Dec 10–12). ACM, New York, NY, 1979, pp. 24–32.
27. Allchin, J. E. and M. S. McKendry. *Synchronization and Recovery of Actions.* Preprint: 2nd ACM SIGACT-SIGOPS Symposium on Priciples of Distributed Computing (Aug. 17–19, 1983).
28. Allchin, James E. and Martin S. McKendry. "Support for Objects and Actions in Clouds: Status Report", Tech. Report Georgia Institute of Technology, Atlanta, GA, Atlanta, GA, 1983.
29. May, D. *OCCAM.* SIGPLAN Notices (April 1983) vol. 18, no. 4, pp. 69–79.
30. Andrews, Gregory R. "*The Distributed Programming Language SR - Mechanisms, Design and Implementation*". Soft. Pract. and Exper. (1982) vol. 12, pp. 719–753.
31. Feldman, Jerome A. *High Level Programming for Distributed Computing.* CACM (June 1979) vol. 22, no. 6, pp. 353–367.
32. Dijkstra, Edsger W. *Guarded Commands, Nondeterminacy and Formal Derivation of Programs.* CACM (Aug. 1975) vol. 18, no. 8, pp. 453–457.
33. Ramamritham, Krithivasan and Robert M. Keller. *Specification of Synchronizing Processes.* IEEE TOSE (Nov. 1983) vol. SE-9, no. 6, pp. 722–733.
34. Horning, J. J., H. C. Lauer, P. M. Melliar-Smith and B. Randell. *A Program Structure for Error Detection and Recovery.* In: LNCS, vol. 16, E. Gelenbe and C. Kaiser, ed. Springer-Verlag, New York, NY, 1974, pp. 171–187.

35. Anderson, T. and P. A. Lee. **Fault Tolerance: Principles and Practice.** Prentice-Hall International, Englewood Cliffs, NJ, 1981.

36. Hoare, C. A. R. *A Model for Communicating Sequential Processes.* In: **On The Construction of Programs**, R. M. McKeag and A. M. McNaughton, ed. Cambridge University Press, Cambridge, UK, 1980, pp. 229–243.

37. Filman, Robert E. and Daniel P. Friedman. **Coordinated Computing: Tools and Techniques for Distributed Software.** McGraw-Hill, New York, NY, 1984.

38. Manna, Z. and A. Pnueli. *Verification of Concurrent Programs: The Temporal Framework.* In: **The Correctness Problem in Computer Science**, R. S. Boyer and J. S. Moore, ed. Academic Press, London, UK, 1983, pp. 215–273.

39. Nelson, B. J. "Remote Procedure Call", Ph. D. Dissertation, Carnegie-mellon University, Pittsburgh, PA, 1981.

# Mediators: A High-Level Language Construct for Distributed Systems

Judith Ellen Grass

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, Illinois

April 1986

# MEDIATORS: A HIGH-LEVEL LANGUAGE CONSTRUCT FOR DISTRIBUTED SYSTEMS

BY

JUDITH ELLEN GRASS

B.S., Georgetown University, 1975
A.M., University of Illinois, 1977
M.S., University of Illinois, 1982

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosopy in Computer Science
in the Graduate College of the
University of Illinois at Urbana–Champaign, 1986

Urbana, Illinois

# MEDIATORS: A HIGH-LEVEL LANGUAGE CONSTRUCT FOR DISTRIBUTED SYSTEMS

Judith Ellen Grass, Ph. D.

Department of Computer Science
University of Illinois at Urbana–Champaign, 1986
Dr. Roy H. Campbell, Advisor

This thesis describes the *mediated object* construct. Mediated objects support synchronization and scheduling for systems programming within distributed systems. Mediated objects are based on a resource view of systems, and fit within a programming methodology that emphasizes resource modularity, synchronization modularity and encapsulated concurrency.

A mediated object consists of an interface specification, a data abstraction construct (an *object*) and a separate *mediator* module that specifies synchronization and scheduling within the mediated object. The mediator displays many interesting features. These include: an adaptation of guarded commands; *keys* that allow requests to be examined and manipulated before they receive service; parallel guard execution; coupled and uncoupled modes of service execution.

The design of the mediated object construct is first presented informally with many programming samples. A temporal logic specification is also presented as a formal description of the construct. The temporal logic may be used for verifying mediated objects. A sample verification is included. Few practical languages have been specified with temporal logic. The specification provided helpful feedback during the development of the construct.

Finally, the thesis discusses a few aspects of implementation and offers suggestions for future research.

Dedicated to my parents:
Luther M. Grass,
G. Kathleen Grass.

# ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Roy H. Campbell for the helpful advise and support he gave me while I was working on this thesis. I would also like to thank my committee: Professors Jane Liu, Geneva Belford, Mehdi Harandi and M. D. Mickunas for their time and effort. Anda Harney deserves a thanks for all the little chores she took care of for me on the way.

Special thanks are due to Professor Simon Kaplan, without whose help I might never have pulled the temporal logic section together. Vipin Swarup also contributed to that effort.

Finally I would like to thank all the people that helped to keep me sane and motivated for the long haul that was graduate school. First of these is Dennis Mancl who got me to believe I could do this hard stuff if I really wanted to, and kept me going when I was ready to pack it in. I am also grateful for Scott McEwan's help. I would also like to thank all the minions of SRG who would lend an ear when I needed it. Final thanks to Penny, who carried me over the rough spots, dropped me only occasionally and gave me lots to think about other than my thesis.

This work was done while the author was supported by NASA grant NSG–1471. This grant funds the EOS project, direct by Professor Roy Campbell.

# TABLE OF CONTENTS

CHAPTER

# CHAPTER 1.

# INTRODUCTION

This thesis introduces the *mediated object* construct for implementing synchronization and scheduling in distributed systems. The mediated object combines a data abstraction module with a synchronization and scheduling module called a *mediator*. The mediated object construct supports systems programming applications that require complex and flexible synchronization and scheduling schemes. The research was prompted by the recognition that many of the existing language constructs either overly constrain concurrency, make expression of some kinds of synchronization and scheduling difficult, or due to formal language design considerations fail to provide practical support for real programmers.

## 1.1. Motivation for Concurrency and Distribution

Computer systems that support concurrency are significantly more complex than simple single–user sequential systems. Managing concurrency also creates a certain amount of overhead. Distributed systems are yet more complex and entail even more overhead. In many applications the benefits of such systems outweigh the costs of developing, implementing and using them. Some applications could not be computerized at all without such support.

### 1.1.1. Concurrency

The first attempts to provide a form of concurrent programming in computer systems occurred during the development of second generation computer systems in the early 1960's[29]. In order to efficiently share access to an expensive mainframe among many users,

*multiprogramming* was developed. In multiprogramming, a single processor switches between different user jobs, keeping the processor as busy as possible[97]. Efficient use of sharable resources has continued to be a prime motivation for developing concurrent programming. The concern for efficient use of resources led to the beginning of *multiprocessing* systems at about the same time[29]. Multiprocessing systems combine several processors in one computer to increase throughput.

An early application of multiprocessing applied separate processors to manage input and output while a main processor managed other computations[107]. This increased throughput by freeing the main processor from driving slow mechanical I/O devices.

Later applications increased the number of main processors contained in one computer [29]. Multiprocessors are used to support greater throughput in time-sharing systems and to speed up individual users' jobs. Single users may exploit multiprocessing either by explicitly partitioning a program to perform separate tasks concurrently, or by allowing the computer to analyze the program for operations that may be done in parallel and to execute the program implicitly in parallel[29]. The former is often referred to as *multitasking.* This is a typical application of *concurrent processing.* The latter approach is commonly used for numerical processing, and is often what is meant by *parallel processing.* Concurrent processing does not usually refer to implicit analytical approaches to parallelism. In this thesis we will address only concurrent processing.

Concurrency may be applied solely to speed up the execution of algorithms that are not inherently concurrent. Concurrent algorithms are sometimes applied to mathematical problems that have adequate sequential algorithms. One example is a concurrent algorithm applied to a matrix multiplication problem[50].

Other problems appear to be inherently concurrent, and a concurrent solution for such problems is both more natural and easier to develop[29]. Many real–time systems are in this category. For example, air traffic control programs that must track a number of airplanes at once may be more easily implemented using the support for dealing with many independent events provided in a concurrent system. The actual implementation of such a system may be made by time–slicing a single processor, or by taking advantage of multiprocessing. In either case, the system itself is best modeled as a concurrent one.

### 1.1.2. Distribution

Distributed systems join a number of processors that, unlike multiprocessors, do not share memory. Instead, in a distributed system many independent, possibly heterogeneous, computers are tied together by communication lines[97]. The motivation for creating such systems is in part the same as that for creating concurrent systems. Distribution can be used to efficiently share resources and to speed up computations. It can also be used to increase system reliability and to implement communications systems.

Resource sharing in distributed systems differs from sharing in simple multiprocessor systems in that resources located at remote sites may be accessed by users that do not have such resources available at their own site. Because of distribution, expensive but not heavily used resources need not be duplicated for every computer in a system. In some distributed systems it is possible to increase the throughput of the entire distributed system by shifting computations from heavily loaded machines to computers that have idle cycles. This kind of load balancing is another way of efficiently using the entire system resource.

System reliability may be improved by using the redundancy present in distributed systems. When a processor in such a system fails, it may be possible to shift its workload to another functioning processor. If a resource located at one location fails, another compatible resource may still be functioning elsewhere.

It is also possible to use the communications network inherent in a distributed system for pure communications applications, such as electronic mail[97].

Real–time control systems can make use of small processors configured in a distributed system to take advantage of the great amount of concurrency possible and the added reliability and efficiency such systems offer.

## 1.2. Language Design Goals for Distributed Systems

Developments in higher–level language design for managing concurrency and distribution has been driven both by the changes in systems architecture and by software engineering developments. As the dominant computer system architecture has changed from uniprocessors to multiprocessor mainframes to networks of uniprocessors and distributed multiprocessor systems, the context for developing concurrent programming tools has also changed. Meanwhile, the cost effectiveness of some synchronization mechanisms has changed as the cost of computer memory and processors has dropped.

Whether for multiprocessor or distributed systems, concurrent programs are extremely difficult to develop. Human beings are not well adapted to dividing their attention between simultaneous tasks, and can only comprehend concurrency with a great deal of effort. Programming is made more difficult by the fact that processes in such a system operate at different and unpredictable rates, and also must cooperate in order to share common

resources or exchange information. Obviously, language and systems tools are required in order to program concurrent applications productively. The nature of these tools has changed with our understanding of the nature of concurrent processing and software engineering.

Chapter two of this thesis discusses in detail some developments in programming language design to support concurrent processing. Concurrent language design has followed closely in the footsteps of sequential language design. Early research developed a number of primitives, including the *test and set* instruction[97], and algorithms using such primitives. Later research has concentrated on using higher–level constructs to ease the programming burden and to ensure more reliable results.

Software engineering research has found several techniques to be useful in managing the complexity of programming large systems of sequential programs. These include modular programming and the principle of "information hiding"[27]. These techniques help to impose organizational structure on programs and make the development of complex programs more manageable. Constructs developed for concurrent programs have adopted this approach as well. Newer constructs manage concurrency through higher–level abstractions that implement modularity and "information hiding".

All language design is the result of trade–offs. Languages that offer a framework for easy verification are often limited in flexible expressive power. Some languages hide many details of communication between distributed processors, making an easy environment for applying distribution to many problems. Others make the user manage the communications protocols directly, providing a more flexible mechanism that is more difficult to use. In most cases, the goal has been to provide tools to structure the development of concurrent, distributed pro-

grams.

The difficulty of developing reliable programs cannot be met by careful language design alone. The best of tools can be badly used. This problem has been addressed by developing rigorous program development methodologies and methods for program verification[41,60]. Both of these efforts are supported by formal language specifications. Many of the tools created for the developing and verifying sequential programs can not be directly applied to concurrent programs. Many of the general techniques of partitioning problems and applying formal logic to the problem of verification have been used.

The mediated object was developed for managing resources in a distributed embedded system. The design emphasizes resource modularity, synchronization modularity and flexibility. A mediated object encapsulates a shared resource that may be accessed by requesting specific services from the mediator. The mediator is a program contained within the mediated object that encapsulates synchronization and scheduling for the object. Concurrent, noninterleaved accesses to the resources are allowed within the mediated object. In this sense, the mediated object allows encapsulated concurrency. The range of synchronization and scheduling constraints that can be specified by the mediator is very broad. An additional feature allows the mediator to manage pools of heterogeneous resources that provide equivalent services.

The mediated object design is supported by a temporal logic specification. Mediators may be verified using this specification and the temporal proof system developed by Manna and Pnueli[83]. We have developed an example of such a verification.

## 1.3. Outline of the Thesis

In the following thesis, we will discuss some existing synchronization mechanisms as they apply to distributed systems. Chapter two examines the literature on concurrency and synchronization in programming languages. Chapter three discusses the design goals for mediated objects. Chapter four presents an informal description of the mediator construct. Chapter five contains a formal temporal logic specification and a sample proof. Chapter six presents some observations on implementing mediators. Chapter seven discusses the conclusions of the thesis and offers some possibilities for further research.

# CHAPTER 2.

# A SURVEY OF THE LITERATURE

In designing a concurrent language, the designer faces all the familiar problems of sequential language design. These include providing sequential control structures and support for building abstractions. A concurrent language design must address three central problems that are not part of sequential language design [8,36]. A concurrent language must have a notation and mechanism to:

- represent concurrency;
- provide interprocess communication;
- synchronize process interactions.

These concurrent design components and sequential design considerations are not completely orthogonal, but it is convenient to look at these as separate issues.

## 2.1. Concurrency

The earliest problem faced in designing concurrent languages and systems was to provide notations and mechanisms for parallel execution. All of the solutions are based on the *process* concept. A process is a sequential program that is in execution[29].

Processes may be used in a number of ways. In a *multiprogramming system*, the processes represent different jobs that may or may not be related. These jobs take turns using a single processor in order to better the overall system performance [97]. Multiprogramming may be implemented on computers that contain a single processor, or on comput-

ers that contain several (*multiprocessors*). This use of processes is essential to time-shared operating systems. A *multiprocessing system* uses multiple processors that either share memory or communicate by a network. While multiprogramming techniques may be used to manage work for a single processor in such a system, this kind of system can support true parallel process execution.

*Coroutines* fit nicely into a multiprogramming view of concurrency. A coroutine is a program whose execution is interleaved on a single processor with the execution of two or more other coroutines. No synchronization is provided beyond the ability to suspend one coroutine and resume execution of another [8].

**Fork** and **join** primitives are low level constructs that allow dynamic creation and termination of concurrent processes. When a **fork** instruction is executed by a process, a second new process is created that will execute concurrently with the process that created it. The original process is often called the *parent process.* **Join** has been specified in several ways[8,97]. A **join** operation combines the execution of two processes into one process. In one implementation, if two processes both execute a **join**, the process that executes the **join** first is terminated. The other process is allowed to continue[97]. In other implementations, only the parent process may execute a **join**. The parent then waits for the child process to terminate before the parent may proceed[8].

The **fork** and **join** construct have great expressive power, and have been widely used in many systems[8]. The problem with this construct is that the **fork** instruction can be used in an undisciplined manner to produce unstructured code. This is the same kind of problem as the **goto** statement creates for structured programming[97].

**Cobegin** and **coend** statements provide a more structured way to dynamically create and terminate processes [31,115]. When a **cobegin** is executed, two or more processes are created that will proceed concurrently. The **cobegin** statement is the beginning of a block that must end with a **coend** statement. All processes spawned by the **cobegin** terminate together when all have reached the execution of the **coend** statement. This is a block structured control statement that supports structured programming[8].

Modern concurrent languages use the **fork/ join** primitive or **cobegin / coend** statements to implement concurrency. Generally routines that may be executed concurrently are explicitly declared as processes. Some languages implement systems of processes that are static. In these, process declarations are equivalent to a **cobegin** that spawns a fixed number of processes that will not grow or diminish during execution. This is true of Brinch Hansen's Distributed Processes [18] and Andrews' SR [6]. In other languages, invocation of a process by a call spawns a process (essentially a **fork**) which will terminate when its sequential execution terminates. Concurrent PASCAL [17] and Path Pascal [67] take this approach. Frequently the spawning of a process is done implicitly, but there are a number of languages that include the **cobegin** construct explicitly. Argus [77], CSP [50], OCCAM [86] and Edison [19] are some examples.

The simple ability to synchronously begin and terminate process execution that is provided by the **cobegin** construct provides only a very weak form of synchronization that is only sufficient if the processes do not cooperate or share variables. Two processes that share variables may interfere if no additional synchronization is implemented. *Interference* refers to the unexpected and undesired interactions that may occur between unsynchronized or

badly synchronized processes. A familiar example to UNIX®[1] users would be the scrambled screen that sometimes results from *forked* background jobs outputting to the same terminal on which a user is editing a file. Similar interference may occur to shared data, with the actions of one process invalidating assumptions about that data on which another process must depend. Additional synchronization tools are needed to prevent this kind of data corruption.

## 2.2. Process Communication and Synchronization

The nature of the synchronization mechanism design depends on the way in which processes may communicate. There are essentially only two ways in which processes may exchange information. Processes may share variables by accessing the same addresses in memory, or they may send and receive messages from each other. The difference is less clear if the messages may contain call by reference parameters, but the distinction is still useful. In many ways, these are two sides of the same coin. Anything that can be done within one communication framework can be done within the other [72], although the elegance of the resultant programs may differ. These communications mechanisms are suited to different environments and present some unique characteristics.

Shared variable communication fits quite naturally into a system supported by a multiprocessor mainframe [36]. It is possible to duplicate the effect of a shared variable system on a network, but this requires either extra wiring or message passing support. In a shared variable system, the user is ultimately responsible for providing all synchronization, as none is inherent in the communication mechanism [8].

---

[1] UNIX is a trademark of Bell Laboratories

The semantics of message passing has implicit synchronization characteristics. A process must send a message before another process may receive it. Implementing a message passing system requires more operating system support than must be provided in a shared memory system. Most of the additional support is in providing communications links between processes [118]. Implementing communications links involves a number of decisions concerning how processes synchronize when they send or receive messages. The most basic decision is whether a sender and receiver must synchronize when they pass a message.

It is possible to roughly group synchronization constructs by the type of interprocess communication used to support them [8,36]. This classification is clearer for some of the original synchronization constructs. It becomes less clear when more recent languages are considered. Because more recent structures for synchronization (such as atomic transactions) represent higher levels of abstraction, it becomes easier to imagine them implemented within either framework.

Synchronization in programming languages is used in two distinct ways. The most fundamental application of synchronization is used to ensure that operations on shared data occur "indivisibly". Processes may also be synchronized to enforce access ordering consistent with higher level abstractions.

### 2.2.1. Atomicity

Operations that appear to act "indivisibly" are frequently referred to as *atomic actions* [12,26]. Some researchers have understood this definition to mean that all atomic actions must exhibit "all–or–nothing" semantics [1,2,76,117]. By this definition, all atomic actions must either complete their intended effect, or have no effect at all. This adds the requirement

that all atomic actions necessarily provide error detection and recovery in addition to safe synchronization. The concept of "indivisible execution" is quite useful apart from considerations of recoverability. This definition has frequently been used in research on verification [69,71]. We will use the term "atomic action" in this sense only.

Atomic actions are frequently implemented to ensure data consistency in the face of concurrent accesses. Concurrent arithmetic on a shared variable illustrates a low–level use of the concept. Consider two processes, P and Q, which execute the following statements concurrently:

P: x := x + 1; Q: x := x + 1;

As users of higher level languages, we tend to think of statements as behaving atomically. If $x$ were equal to 1 before these operations occurred, we would expect the only possible final value of $x$ to be 3. However, these statements would be translated by a compiler into code that reads the value of $x$ into a register, adds 1 to that value and stores the results back into $x$. Since P and Q may interleave these steps in various ways, the actual result could be 1, 2 or 3 [8]. Synchronization to ensure data consistency and non–interference by enforcing atomicity must be applied on this rather low level and on higher levels of abstraction as well. A database "atomic transaction" is one such case, in which special effort must be taken to ensure accesses to related groups of data maintain the consistency of the entire database [36,40].

### 2.2.2. Conditional Synchronization

There is a second application of synchronization that is not concerned with maintaining the indivisibility of state changes implied by atomicity, but rather with limiting which state changes will be allowed. Typically, such synchronization has been used to ensure that the invariants of a data abstraction are maintained [4,47–50,70,92,93]. Implementations of concurrent buffer objects uses synchronization in this way. Operations on such an object must be synchronized so that no operation will be allowed to "read" data from an empty buffer or to "write" data to a full one. In this case, synchronization is not enforced solely to prevent concurrent accesses from interfering. Whether or not access will be allowed is based on the condition of the shared resource and on the type of operation. Such synchronization is frequently called *conditional synchronization* [8].

## 2.3. Shared Variable Synchronization

A number of synchronization primitives have been developed for concurrent systems using shared variables for communication. The main applications of these primitives are to ensure mutual exclusion when the shared variables are accessed and to implement conditional synchronization[8].

### 2.3.1. Busy–Wait

Initial approaches to synchronization in shared variable systems were rather low–level constructs. One of the earliest of these was the *busy-wait* loop [31]. In this approach, critical sections (code acting on shared data) were protected by shared Boolean variables that had to be true for a process to enter the section. If the variable was false when tested, the testing

process would continue testing it repeatedly until it became true (through the action of some other process). This solution had the virtue of not requiring special hardware support, but suffered from many drawbacks. This solution wastes a good deal of processor time, an especially serious problem when processors were very expensive. Another serious problem was the difficulty of designing and implementing safe synchronization with such low–level tools.

### 2.3.2. Semaphores

Semaphores [31] are a higher level construct than busy–waiting for synchronization. **P** and **V** operations on a semaphore implement a test on a Boolean. Executing a **P** operation causes a process to release the processor if it is not immediately able to execute. Another process leaving its critical section executes a **V** operation that will reactivate a process waiting on a corresponding **P**. Simple semaphores and counting semaphores can be combined to implement a variety of conditional synchronization schemes as well.

The actual implementation of the semaphore may be done with a busy–wait, but this may be avoided by including these instructions as part of the operating system nucleus[29]. In this case processes can be suspended and reawakened on **P** and **V** operations by operations that the nucleus must have for handling queues of processes.

Although they are a higher level primitive than the busy–wait, semaphores are not a structured programming construct. Subtle errors in using these operations can lead to serious synchronization errors. The programmer is responsible for ensuring the every *P* operation is balanced out by a corresponding *V* operation. In a large program the individual operations implementing synchronization are not localized, making it hard to find all the instructions used to implement a certain constraint. Programs written with semaphores are

often hard to develop, hard to understand and hard to maintain.

### 2.3.3. Conditional Critical Regions

Once some low–level means of synchronizing processes were found, higher–level language constructs that would allow a more structured approach to synchronization began to be developed. *Conditional critical regions* [15,16,48] presented one successful approach. In a conditional critical region shared variables are confined to a construct called a *resource*. Such variables may only be accessed in mutual exclusion. Sections of code that access a resource are identified as a *region*. Statements that are enclosed in a region may be accessed by only one process at a time. Conditional synchronization can be implemented by including a Boolean test at the entry of the region.

This approach does not solve all of the problems associated with the low–level, unstructured nature of semaphores. The tests and manipulations of Boolean conditions used to implement conditional synchronization are once more scattered throughout the code. This makes it difficult to read the code and understand the exact nature of constraints. Despite the difficulty of structuring the use of conditional critical regions, this construct still has a certain attraction. Brinch Hansen chose to use a simplified version of critical sections as the main synchronization construct in Edison [19].

### 2.3.4. Monitors

A *Monitor* is a higher–level language construct that ensures that a data resource will be used in mutual exclusion, without requiring the programmer to explicitly program low–level synchronization[16,31,49]. The syntax of a monitor is similar to that of a *class* (figure

2.1)[13]. The monitor contains encapsulated data and a number of operations defined on the data. It functions as an abstract data type. As in an abstract data type, the only access to encapsulated data is through the operations defined on them. Only one process may be active in the monitor at a time.

```
class readers and writers: monitor
 begin  readercount: integer;
      busy: Boolean;
      OKtoread, OKtowrite: condition;
   procedure startread;
      begin if busy ∨ OKtowrite.queue then OKtoread.wait;
        readercount := readercount + 1;
        OKtoread.signal;
        comment Once one reader can start, they all can;
      end startread;
   procedure endread;
      begin readercount := readercount - 1;
        if readercount = 0 then OKtowrite.signal
      end endread;
   procedure startwrite;
      begin
        if readercount ≠ 0 ∨ busy then OKtowrite.wait
        busy := true
      end startwrite;
   procedure endwrite;
      begin busy := false;
        if OKtoread.queue
           then OKtoread.signal
           else OKtowrite.signal
      end endwrite;
 readercount := 0;
 busy := false;
end readers and writers;
```

Figure 2.1. Readers and Writers Monitor[49].

This simple view of a monitor does not provide for conditional synchronization. There are many variations of monitors that use different schemes to provide conditional synchronization. Hoare's scheme [49] (figure 2.1) parallels the solution used in conditional critical regions. In this approach, special "conditional" variables are defined in the monitor. Processes within the monitor may execute the operations **wait** and **signal** on them. When a process executes a **wait**, it becomes blocked and relinquishes control of the monitor. When another process executes a **signal** it will awaken a blocked process and suspend itself until no more processes are blocked on **waits**. Many processes may be blocked by the monitor, but at most one will be executing. This approach has been combined with priorities to add a simple scheduling capacity to monitors.

Other conditional synchronization schemes for monitors appear to be variations on this basic theme. Concurrent Pascal [17], uses queue variables with **delay** and **continue** operations that implement a slightly less powerful scheme, in that not all monitors written in Brinch Hansen's scheme may be translated directly into Hoare's scheme without adding additional routines to the monitor[56]. The **continue** statement, however, is somewhat less costly to code than the **signal** statement. The **continue** statement causes the process that called it to return from the mediator. **Signal** is more difficult to implement, as it must suspend the signaling process and ensure that that process will not become unblocked before all processes blocked on **wait** have resumed[8].

The use of special Boolean condition variables tends to separate the conditional synchronization testing from the conditions that lead to a process blocking or resuming execution. The result can be inscrutable code unless much care is taken. Hoare [49] proposed a conditional **wait** that would operate on an arbitrary Boolean expression. In this scheme,

whenever a process exits a monitor an implicit **signal** is executed that causes waiting processes to retest their conditions. This is somewhat less efficient than the above approaches, but a lot easier to use [8]. This by no means exhausts the variants that have been implemented to deal with conditional synchronization in monitors [36].

Monitors effectively ensure that the execution of monitor operations interleave. For the portion of the operation that is interleaved, no interference may occur. The execution of an operation may be suspended by performing a **wait** or **signal**. This creates a possibility of interference in conditional synchronization because the actions of other processes will be interleaved between the time a certain process becomes blocked and when it resumes operation. For example, a process executing an operation in the mediator may establish a certain condition, execute a **signal** and become suspended. If it requires that condition to be true when it resumes operation, either all possible interleaved executions must ensure that condition is maintained, or the suspended process will need to test for that condition when it resumes[8,119]. Ensuring that conditions will not change for suspended processes like the one described above and writing additional tests to ensure conditions on resumption are not easy tasks.

Systems that include a large number of components may be built using a large number of monitors to manage access to those components. In these cases is may become necessary for a process executing within a monitor to call another monitor. Such calls are referred to as *nested monitor calls.* Nested monitor calls are a problem in the simple monitor scheme presented here[8,97]. Consider the situation in which a process executing in monitor A calls an operation in monitor B. While executing in monitor B, the process still retains mutual exclusion in monitor A. No other process can gain access to monitor A. This will continue to

be so as long as the process has not returned from B. If the process becomes blocked in B, a considerable delay may occur before the process returns from its call. This situation can seriously degrade the performance of a system[8]. In this simple example, only two monitors are involved. It is possible for a chain of calls involving several monitors to tie up all those monitors when the last call made becomes blocked in another monitor. The performance cost becomes correspondingly larger.

Many solutions have been proposed to deal with nested monitor calls. Many of these are cited in [8]. Some have suggested that such calls be prohibited or limited to cases in which the monitors involved were lexically nested (as in Modula[119]). Others have proposed mechanisms to allow a process blocked in a nested mediator call to release mutual exclusion of all its monitors, and to reacquire mutual exclusion when the call becomes unblocked. A final solution is to provide other mechanisms to handle the situations in which nested monitor calls would be used.

The biggest remaining objection to the monitor scheme is that monitors provide only mutually exclusive access to the resources that they encapsulate. Many resources may require higher degrees of parallelism to be used most efficiently. Certain kinds of accesses to shared data do not pose threats to data consistency. For example, several processes reading from a table simultaneously do not pose a threat of interference. This kind of simultaneous access cannot be constructed within a monitor. Solutions to the *readers and writers problem*[27] implemented with monitors use a monitor like that in figure 2.1 to apply a protocol, but the actual read or write operation must occur outside of the monitor. However, the monitor cannot enforce the use of synchronization control. A process may access the shared data without it. This may simply damage the shared resource, or it may deadlock the monitor

[115].

Monitors have proved to be an enormously influential mechanism for synchronization. Many languages have been extended for concurrent uses by adding some variant of the monitor concept. Many of these operate in shared memory systems: Concurrent Pascal [17], Mesa [87], Modula [119], Concurrent Euclid [54] and Path Pascal [67]. The influence of monitors has also been felt in languages based on message passing: DP [18], Ada™[2] SR [6,7], Distributed Path Pascal [66] and Argus [117].

## 2.4. Open Path Expressions

In monitors, mutual exclusion synchronization is provided implicitly by the construct, while conditional synchronization is done with user programmed **signal** and **wait** statements that are scattered throughout the module. An *Open Path Expression* is a notation based on regular expressions that can specify many complex synchronization constraints in a single expression[21,25]. Open path expressions have been introduced into an encapsulated data module called an *object* to manage concurrency in Path Pascal and Distributed Path Pascal [24,66,67]. As in a monitor, processes may access data only through the operations the object defines on that data. However, access is not limited to mutual exclusion. A path expressions may specify that an unlimited number of processes may execute an operation simultaneously, that some fixed number may execute it, or that execution may occur only in mutual exclusion. Similarly, certain combinations of operations may be allowed to execute together, while other combinations are prohibited. In all cases, the basic unit of synchronization is the operation.

---

[2] Ada is a trademark of the U. S. Government, ADA Joint Program Office.

A path expression specifies access restrictions using lists of operation names and path operators. This has been called an "operational approach" to synchronization [8] because synchronization is expressed as allowable sequences of operations on an object. There is no mechanism provided to enforce synchronization based on the value of variables encapsulated within the object. This leads to a certain awkwardness when implementing conditional synchronization within Path Pascal objects. It is possible to do so using nested objects to implement what is essentially a monitor-like signal/ wait protocol. Scheduling, a type of conditional synchronization, can be implemented by similar methods.

There have been some attempts to extend path expressions to include tests of Boolean conditions. Predicate path expressions are one such extension[4]. David Mizell further generalized path expressions to address the conditional synchronization problem[88]. Mizell's generalized paths have lost their regular expression form and have developed into a programming language including loop and branch constructs and local variables.

Path expressions offer a very expressive notation for a broad range of synchronization schemes. The programmer is not limited to mutually exclusive execution, so it is possible to build customized synchronization based on the semantics of the operations. Path expressions also effectively separate the design of synchronization from the design of operations for many problems. The separation is less effective when nested objects must be used to implement conditional synchronization. However, nested objects do assure that all uses of the resources they implement will be properly synchronized. Finally, path expressions take the difficult chore of writing low level synchronization code out of the hands of the programmer, resulting in safer code from the start.

The Path Pascal object has proved to be very flexible. It has served as a host language for implementing real-time deadline processing mechanisms [116], extensions for fault-tolerant computing [55,105] and software capabilities [79].

Path expressions have been used in a number of projects related to systems development and software engineering. The Clouds project at the Georgia Institute of technology has used path expressions [2]. They have been adopted as part of the SLAN-4 specification language [9] and have played a part in other research on specification and verification [11,90,108].

## 2.5. Synchronization for Message Passing Systems

Loosely coupled processors in a distributed system inevitably communicate by some form of message passing. The act of sending and receiving a message can become the basis for synchronization because it has an inherent order. In such a system, the contents of the message takes the place of shared variables. The major problems in designing a concurrency scheme based on message passing are identifying the source and destination processes for a message and designing a precise semantics for synchronization.

### 2.5.1. Naming

One simple approach to naming in a message passing system is to require the sender to name its intended receiver, and the receiver to directly name its intended source. In general, the *direct naming* approach is easy to implement and easy to understand[8]. Direct naming, however, is somewhat limited in the kinds of relationships that it can easily express.

*Server* processes are not easily implemented with direct naming. A server process is meant to provide a service to whatever *client* process calls it. A disk driver routine is an

example of such a process, as many processes will call on it to read files for them. The server processes must be flexible enough to serve any client that calls on it. It cannot reasonably anticipate the names of all potential clients [8,36,115]. On the other hand, some *pipelines* are fairly easy to program in a direct naming framework. In a pipeline, one process produces a stream of output that immediately becomes the input of another concurrent process. A pipeline may consist of several stages, each executed by a separate concurrent process. In cases where a given static ordering of pipeline processes will be maintained, the direct naming scheme can easily be used. However, pipelines that are dynamically created at run–time can not easily be implemented with static naming schemes[8]. This approach would make it difficult to implement pipelining as it occurs in UNIX™[101].

CSP uses a static direct naming scheme[50]. In CSP, the names of interacting processes occur as constants, thus all the names of processes that may interact must be know when the system is compiled. Because the naming in CSP is entirely static, certain applications are difficult to implement using CSP.

One alternative to direct naming allows processes communicate only indirectly through a known intermediate, often called a *channel*. Channels may be named statically, fixing the names at compile time, or they may be computed dynamically at run time. Static channel naming suffers from the same kind of inflexibility as direct naming[8].

Channels may be implemented in many ways. In a *mailbox* scheme, all processes share a list of intermediate locations to which each may send messages, and from which each may receive messages. This scheme may be quite expensive to implement without special networking support[8].

A more restrictive approach that is easier to implement ties the intermediate location to one receiver. In this case it is called a *port*. Ports allow multiple client/ single server relationships to be easily implemented[8]. Many languages, including Ada [28], have been implemented using ports. Ports have been proposed as an extension to CSP as well [20,65] and implemented in the CSP–like language OCCAM [86].

### 2.5.2. Synchronization

The kind of synchronization provided by a message based language is determined to a large extent by the kind of synchronization enforced between senders and receivers [106]. Communication between a sender and a receiver may occur either synchronously or asynchronously. When a sender and receiver communicate synchronously, sending and receiving must occur together. If a process is ready to send, and its intended receiver is not ready to receive, the sender must wait for the receiver. If the receiver is ready to accept a message, and the sender is not ready to send it, the receiver waits for the sender. The execution of either a **send** or a **receive** statement may cause a delay. When a sender and receiver communicate asynchronously, there must be a buffer between the sender and receiver. The sender process is not delayed if the receiver is not ready. It just transmits its message and continues on. The message is stored in a buffer until the receiver is ready to accept it by executing a **receive**. If a **receive** is executed when no messages are waiting, the receiver usually must wait for a new message. In some cases, languages provide a non–blocking **receive** statement to use as a test for pending messages.

Either of these forms of communications provides a basic measure of synchronization between the sender and the receiver. In either case the receiver can be assumed to be execut-

ing either "behind" or "in step" with the sender. When these communications primitives are combined with *guarded commands* [32], communications can be controlled by the internal process state. This facilitates implementing conditional synchronization.

A guarded command consists of a Boolean expression followed by a statement or statement list. These are built into a list of alternates as in figure 2.2. When a guarded command is executed, some statement with a guard that evaluates to *true* will be executed. If no guard evaluates to *true*, the command aborts. Guarded commands are also used in a looping construct in which failure to find a *true* guard causes the iteration of the statement to terminate.

In CSP, guards may contain *receive* statements[50]. If no message is ready to be received, the execution of the statement list is delayed until some message arrives. CSP has been extended to allow synchronous *send* statements in guards as well [20,65,109]. This is a good deal more difficult to implement because it requires some protocols to be developed to arbitrate between pairs of processes wishing to communicate[8].

```
if G1 → S1
□ G2 → S2
□ G3 → S3
        ...
□ Gn → Sn
fi
```

Figure 2.2. A Guarded Command[32].

### 2.5.3. Remote Procedure Calls

In general, the **send** and **receive** primitives discussed above operate on a rather low level. Often it requires more than one exchange of messages to program an interaction between processes[8,36,115]. For instance, in a client/ server interaction two sets of message must be exchanged. The first exchange requests the service. The second one is implemented to obtain the results of that service. This type of relationship is better served by a traditional procedure call mechanism, where waiting for the results is implicit in calling for the service. However, the traditional procedure call mechanism is not suited to distributed computing.

The *remote procedure call*[89] was developed to provide this kind of higher level synchronization over a network. The request for a service is expressed as a call, with parameters, to a remote procedure. The remote procedure is executed by the server process as a "proxy" for the client process. On completion of the remote process, the client receives the results of the computation and continues executing. A remote procedure call implements message passing at service initiation and conclusion that otherwise the user would have to provide.

There are many possible ways to manage server processes in a remote procedure call scheme. The remote procedure may be declared as a simple procedure, but implemented as a process that continuously loops, waiting for a call to begin execution. Remote procedures are implemented this way in SR [6,7], and in Distributed Path Pascal [66]. SR has other constructs that allow concurrent execution within a server object, as remote calls do not. DPP implements concurrent execution by spawning processes as they are needed within the remote object. This is transparent to the user. Argus guardians implement concurrent execution in

a similar manner[77].

Remote processes may implemented by specific statements for explicitly accepting calls from clients. Ada *tasks* [28,42,114,122] use such **accept** statements in guarded commands to initiate remote services. The Ada remote call mechanism is called a *rendezvous*. Brinch Hansen's DP [18] takes a similar approach. In these languages, execution of a remote procedure by a server process occurs in mutual exclusion. In SR[6] a similar construct allows multiple processes to execute. The user must explicitly declare one process for each potential concurrent process execution. Each server process is tied permanently to one client. This is a rather inflexible notation, but does allow simultaneous service to multiple clients.

Remote procedure calls have been used to implement *atomic transactions* in the Argus [77] and Clouds [2] systems. They are used to implement an *at most once* semantics for the remote call[8], and to ensure that either an action will be completed once, or it will have no effect at all. In this case, the remote call implements both synchronization and failure atomicity. The amount of concurrency possible in such systems is severely constrained in the interest of providing an environment for recovery.

### 2.5.4. Remaining Interference Problems

The use of synchronous or asynchronous message passing does not in itself guarantee that processes will not interfere [5,91,103,113,115]. Some kinds of interference are eliminated because the addresses that processes access in this scheme are disjoint. However, in order for communications between processes to be useful, there must be assumptions made by and about both partners in the communications. These assumptions are made on the basis of the process' own state and the communications received from the other process. It is possible for

the state of a sender process to change between the time a message is sent and the time it is received. If the message has been invalidated by such a state change, the receiver may act under invalid assumptions. Asynchronous message passing systems are prone to this problem, and special protocols generally are implemented to deal with it. The delay in transmission during which a message may become invalid does not exist in synchronous systems, yet interference may still occur. Once a message has been sent and received, its contents may still be invalidated by the continued execution of the participants [8,104]. The programmer must prevent these problems using the basic synchronization tools provided.

## 2.6. Scheduling

Any processing module that can not service all requests as they are received must implement some form of scheduling to choose between pending requests. Scheduling is used to decide which blocked process will be allowed to continue executing when a resource becomes free. Many components of operating systems are scheduled in one way or another in order to provide efficient service. When time is a consideration, efficient scheduling is important. For instance, it is required as an inherent part of any scheme using priorities or deadlines to ensure timely service [29].

Very few languages implement scheduling primitives for users. In most cases, users are left to build their own schedulers out of other language primitives. As T. Wei mentions in his thesis [116], any concurrent language must implement scheduling on the execution of parallel processes. This is frequently done implicitly, as in Path Pascal, Concurrent Pascal, DP, and Argus [36]. Some monitor schemes have been implemented that allow a priority to be added to the **wait** statement [49]. HAL/S, a language for real-time programming [57] (cited in

[116]) implements dynamic scheduling in a complex framework. Ada implements a static process priority scheme used for preemptive scheduling. Maintaining consistency in the face of preemption is a problem for the user to solve.

Andrews' SR language[6] includes a scheduling primitive. Once an operation invoked by remote procedure call has been accepted, the choice of which caller will be allowed to execute is determined by a scheduling parameter in the **accept** statement. This may refer to a single parameter of the call, or to a function based on the call's parameters. Andrews claims that the implementation is somewhat expensive because the scheduling requires reevaluation each time a call selection is done, but that it is not more expensive than user coded solutions [6,7]. Dennis Leinbaugh [74,75] has also integrated scheduling into a concurrent language design. The language it is based on is similar to PL/I, and the notation is verbose and strongly compartmentalized.

It is not always possible to reconcile a synchronization construct with such scheduling mechanisms. In mechanisms that synchronize on the basis of a class of requests (such as all requests for a "read" operation), once such an operation has been enabled, a scheduling primitive may chose a particular request to execute. In a synchronization system that allows the characteristics of an individual request to determine its eligibility, such a scheduling primitive is not workable.

In real–time programming, the separation makes it easier to estimate the execution time of operations during compile time [116]. The ability to estimate execution time is essential to programming in real–time [3,34,36,73,97,120], separating synchronization from timing makes at least part of the determination static.

Few constructs provide synchronization modularity. Among those are Path Pascal (PP and DPP) [23], sentinel processes [98], and serializers [44]. Serializers are implemented in a LISP environment. Sentinel processes appear to be the imperative language analog. Both combine built–in counters with queueing primitives to allow modular specification of synchronization. These constructs appear to be well suited to FIFO scheduling problems and variants of the reader/writer problem, but are less flexible than desired [14].

Path Pascal encapsulates most synchronization specifications in a path expression. This often provides a high degree of synchronization modularity. The synchronization modularity is lost when conditional synchronization or scheduling is specified. These must be programmed using nested objects. This results in loss of modularity as well as inefficiency due to the implicit scheduling applied at each level of nesting. In order to maintain synchronization modularity, synchronization data must be encapsulated.

Mediated objects provide both data modularity and synchronization modularity and maintain the expressiveness of less modular synchronization mechanisms.

### 3.2. Expressiveness

The expressiveness of a synchronization construct frequently has been demonstrated by programming a number of familiar test problems. The *readers/ writers problem*[27], the *dining philosophers problem* [31] and a simple *ring–buffer* are some examples [36,97]. Test


PRECEDING PAGE BLANK NOT FILMED

problems illustrate synchronization based on different kinds of information. Toby Bloom [14] gives the following categorization of synchronization constraints:

- the operation requested;
- the time of the request;
- the request parameters;
- the resource state;
- the history of events in the resource.

These apply both for providing mutual exclusion and conditional synchronization, and as a basis for scheduling. The expressiveness of a synchronization construct may be measured by its ability to deal with each of these types of information and combinations of these types.

A second consideration in expressiveness is the degree of control over synchronization and concurrency a construct gives to the programmer. Considerations of safety and expressive power often must be traded off in designing practical languages (see Brinch Hansen's discussion of the trade-offs made in Edison [19] ). It is possible to conceive of a construct that provides synchronization based on all of the types of information above, but allows only mutually exclusive access to a resource. Such a construct would not be appropriate for embedded distributed system use [22,73].

Mediated objects allow synchronization and scheduling to be specified using all the kinds of constraints mentioned in list above. The system programmer is given a great deal of latitude in the kinds of synchronization that may be implemented.

### 3.3. Ease of Use

Measuring the "ease of use" of a construct can be very subjective. Programmers develop certain habits when using particular languages that may cause differences of opinion about what is and is not easy to use. This warning aside, some assertions can be made about

ease of use.

The complexity of expressing a certain kind of synchronization should be proportional to the complexity of the abstract synchronization. The notation should be readable. A good rough estimate of usability can be obtained by making a small modification in a synchronization specification and observing the degree of change reflected in the new implementation [14,34]. Ideally, such changes should entail only small modifications in the implementation.

Mediated objects because of their flexibility make a wide range of applications easier to implement than would be possible with more limited constructs. Some simple applications may appear overly complex in a mediated implementation. Some of the wordiness of mediators is meant to make a complex notation more familiar through use of self–explanatory key words. However, there is always a certain trade–off between expressive power and ease of use.

### 3.4. Support for Verification

A useful synchronization mechanism should support program verification. Program verification is a somewhat controversial issue in software engineering [5,30,112]. One group claims that the complexity of proving large programs makes verification either impossible or useless [30]. At the other extreme are claims that formal proof techniques are the only way to ensure that large programs do what they are supposed to do [10,33,37,68,69]. It is not reasonable to expect that verification alone can give programmers total confidence in the programs they design [102]; however, verification supported by careful language design and automated tools certainly should increase the designer's confidence.

Verification considerations are not entirely separate from the features discussed above. Modular programming concepts allow program proofs to be done in a piecewise manner, which reduces proof complexity [37,43,45,46,68,69]. The expressiveness of a programming language affects verifiability as well, but not always in a positive way. Some very powerful programming constructs, such as the *goto* or pointer make some programs extremely difficult to verify [51,78]. Verification can be a guideline in developing such constructs and in their application [5].

A programming language designed to support verification must be precisely specified. This affects both ease of use and portability. Because we do not intend to develop an entire programming language, the "host language" chosen for our synchronization construct should verifiable.

The design of mediated objects has been guided by verification considerations. We have developed a temporal logic specification of mediators that may be used as a base for verification.

# CHAPTER 4.

# THE MEDIATED OBJECT CONSTRUCT

The *mediated object* paradigm is based on object-oriented language design for operating systems applications. In this model, resources are encapsulated and access to them is allowed only through exported operations. The synchronization schemes used in DP [18], Monitors [49], SR[6] and Ada [28] all are examples of languages using this paradigm. The mediated object encapsulates data and allows access to that data through a well-defined interface. Client processes request a service from an exported list of service names, and the mediator determines how the service will be provided. Synchronization and scheduling constraints are specified by the mediator body, and isolated from the definition of data and operations.

The main features of a mediated object are given below.

1) Initialization and termination blocks are included both for the data resource and for the mediator.

2) The essential control structure within the mediator is an adaptation of Dijkstra's guarded commands [32]. Our adaptation uses *delay semantics* [18] rather than Dijkstra's *abort* semantics.

3) Requests are associated with unique *keys* that allow the mediator to manipulate requests and implement scheduling.

4) Guards may contain *status tests* to inquire about pending requests, and Boolean tests which may refer to data contained in pending requests [35,50].

5) The mediator controls execution of client requests by commands allowing coupled and uncoupled client process execution [99]. There is an explicit command to return results to a

client.

6) *Parallel guards* are used to multi–program the mediator. Mediator execution is guaranteed mutually exclusive between guard evaluations.

7) Mediators map the name of a service requested by the client onto that of an appropriate operation. Clients do not call on services directly.

The descriptions that follow first present a schematic of a portion of the mediated object syntax and then an informal semantic description with examples. Portions of these results have appeared in[38]. Chapter 5 presents a formal temporal logic specification of the construct.

The mediated object is one component of a larger language. This thesis does not present a complete language. For our purposes, we assume the "host" language is similar to Pascal. As a result, our mediator syntax is Pascal–like. If this construct were implemented in another language, the syntax would necessarily be quite different.

## 4.1. The Mediated Object

The mediated object includes the definition of encapsulated data and operations defined on that data as well as the specification of the mediator itself. Figure 4.1 is a schema of a mediated object.

A mediated object is made up of three parts: 1) the interface, 2) the encapsulated resource and 3) the mediator. The resource constants, types and variables defined within the object are shared by the resource routines. The mediator maps requests for services listed in the interface onto appropriate operations and synchronizes access. The mediator may contain its own data and routines not accessible to any external caller. Mediator data usually

```
identifier = object
    interface declaration

    resource variables
    resource operations

    mediator
        mediator variables
        mediator routines

        initialization block
        mediator body
        termination block
    end mediator
end object
```

Figure 4.1. Mediated Object Schematic.

consists of flags and counters, although it may also include queue structures for scheduling.

The mediated object is a type, and a user may create several instantiations of a given object. The mediator initiation code is executed when an object is instantiated. The termination code executes when the body of the mediator terminates.

Figure 4.2 presents a complete mediated object. In other examples, only the mediator will be presented. Figure 4.2 contains many notations that have not yet been explained. It illustrates the declaration of an interface, object data (*RW_data*), resource routines (*read* and *write*), and local mediator data (*reader_count*). Object parameters are passed by value and by value–result. Reference parameters seriously compromise data encapsulation and are impractical for current distributed implementations.

Clients request a mediator service which is named in the interface by including the name of the service as a parameter to a call on the object. Once a client process has requested a service, the client is blocked until the mediator returns the results of the completed service.

```
reader_writer = object
   interface
      job : export part
         pid : key client_process_id;
         case service : (read, write) of
            read : (readprm: var some_type);
            write : (writeprm :        some_type);
         end case;
      end export part;

   var  RW_data: some-type;

   procedure read (readprm: var some_type);
        begin  readprm := RW_data  end procedure;

   procedure write (writeprm : some_type);
        begin  RW_data := writeprm  end procedure;

   mediator
        var
           reader_count : integer;
           i, j         : client_process_id;
        init  reader_count := 0 end init
        body
           any i in key: cycle
                   req(i); job(i).service = write ->
                        cycle
                           reader_count = 0 ->
                                exec(i, write (job(i).writeprm));
                                release(i);
                        until true
                   □
                   req(i); job(i).service = read ->
                        reader_count := reader_count + 1;
                        spawn(i, read (job(i). readprm));
                until false
           //
           any i in key: cycle
                   term(i); job(i).service = read ->
                        reader_count := reader_count - 1;
                        release(i);
                until false
        end body
   end mediator
end object
```

Figure 4.2. Reader_Writer Object.

The actual execution of a requested service may be delayed by the mediator. The semantics of a call on a mediator is the same, whether the mediator is installed at a remote location or locally.

## 4.2. The Interface

The interface declaration lists the services provided by the mediated object and the parameters of a request for each service. This provides an external view of the object. Within the mediator, every request that is either pending or being serviced by the mediator is represented by a job descriptor of this form. The descriptor for a specific request is distinguished by the *key* parameter. The syntax we have used is similar to that of Pascal variant records. The name of a requested service serves as the variant record tag. A schematic of the interface declaration is presented in figure 4.3. Figure 4.2 contains a complete example of an interface declaration and its use. The fixed parameters in the interface declaration are those parameters present in all calls to the mediator. In every case, this will include a parameter designated as key by the keyword **key**. The key parameter must be unique for every client that requests service from the mediator. The key is used to identify the client making a request and to manipulate job descriptors within the mediator. The use of keys is described in more detail below.

The *service_list* in the schematic represents a list of services that the mediated object provides. These do not necessarily correspond to the names of encapsulated routines. The mediator maps a request for a particular service onto a specific routine that can provide that service. In some case, more than one routine may be available that provide equivalent services. This allows the mediator to control pools of heterogeneous, but equivalent resources.

```
interface
        identifier : export part
                fixed_parameters
                case identifier : (service_list) of
                        variant_parameters
                end case
        end export part
```

Figure 4.3. Interface Schematic.

Parameters are identified as value parameters and value–result parameters within the interface. The mediator can inspect the value of these parameters to implement appropriate synchronization and scheduling.

A request for a *read* action in the reader_writer example (figure 4.2) takes this form:

```
reader_writer (me, reader, myprm);
```

where *me* is a constant containing the client process' uniquely assigned identifier, *reader* is the name of the requested service and *myprm* is a parameter. This request is mapped onto a mediator job descriptor with the values: job(me). pid = me, job(me). service = reader, job(me). readprm = myprm.

```
        identifier : local part
                fixed_fields
                case identifier : (service_list) of
                        variant_fields
                end case
        end local part
```

Figure 4.4. Local Descriptor Schematic.

The interface definition includes all parameters that pass between the client and the mediator and defines a job descriptor. In some cases it is useful to add additional information to this descriptor for use within the mediator as in figure 4.11. Figure 4.4 shows a schematic for a local extension to the interface. The data contained in the extension is part of a job descriptor accessed using the key. The extension of the descriptor is local data in the mediator.

### 4.3. Basic Mediator Statements

The mediator is composed of several kinds of basic statements and a specialized control structure. The simple statements that can be used within the mediator include: assignments, local mediator routine calls, and the commands **skip**, **exec**, **spawn** and **release**. **Exec**, **spawn** and **release** are statements to initiate services for clients and to return the results of services. These have a key variable parameter that uniquely identifies the client for which the action was taken. This use of keys is explained in detail below. The second parameter of an **exec** or **spawn** statement is a resource operation call. **Exec** permits *coupled* execution of a resource operation (on behalf of a client identified by the key). The mediator initiates a process to execute the operation, and then blocks until the operation has terminated. For example, in the *reader_writer* object above, the statement *exec(i, write (job(i). writeprm));* initiates a *write* operation for client *i*. The mediator blocks until the operation has completed. On the other hand, **spawn** initiates an operation and allows *uncoupled* execution. The mediator does not wait for the operation to terminate, and continues executing mediator code. In the *reader_writer* object, the statement *spawn(i, read (job(i). readprm);* initiates a *read* operation for client *i*.

The **release** command returns the results of an operation to the client and removes the request from the mediator. This may be invoked only after an **exec** has been completed, or a status test (**term**, see below) reveals that a spawned request has terminated. *Reader_writer* (figure 4.2) contains examples of **release** both after coupled and uncoupled service. The separate termination test allows synchronization data to be maintained as services complete. **Release** also makes it possible to delay and synchronize termination and the return of results.

## 4.4. Guarded Commands

Sequences of actions within the mediator body are specified by the control structures presented here, and by parallel guarded commands, which are presented below. The basic mediator control structure is a guarded command as shown in figure 4.5. The prefix **any ... key:** is optional.

The mediator guarded command has many similarities to Hoare's CSP guarded commands [50], which in turn can be credited to Dijkstra [32]. The chosen keywords and semantics are closer to the guarded regions of Brinch Hansen's DP [18]. The concept of *key* is

```
any identifier in key:
  cycle
    guard  -> statement_list;
    □
     ...
    □
    guard  -> statement_list
  until exit_condition;
```

Figure 4.5. Guarded Command Schematic.

related to Hoare's guard command range [52], and to message keys in PLITS [35,36]. The similarities and differences will be discussed below.

A guarded command is a control statement in which different statement lists are chosen for execution based on the truth value of the associated guards. Because the evaluation of guards is central to this construct, they will be explained first. The guarded command will be described after. The application of keys to guarded commands will be presented last.

Guards are made up of a *status test* and Boolean equations. Mediator guard evaluation always results in either a *true* or a *false* value. The special guard **otherwise** is *true* only when all the other guards in the guard command are *false*.

*Status tests* allow inquiries about pending requests for mediator service. These are tests for requests to initiate an operation (*req*) or to return results after the operation has completed (*term*). For the guard *req( i )* to be true, the list of unserved requests must contain a request from client *i*. Once the guard has been *fired* (it's associated statement list chosen to execute), *req( i )* cannot become true again until the service has been completed and the results returned (by **release**( *i* )). The guard *term( i )* is similar, becoming true when the execution of an operation for client *i* terminates.

A Boolean guard paired with a status test may examine the value of a client's request parameters. Each client's request is represented within the mediator by a job descriptor defined by the interface declaration. The descriptor is a variant record containing fields for a key variable, the name of the service requested and the parameters for that service. The service field serves as a tag for variant parameter fields. The descriptor is accessed using the key by indexing on the variable *job*, as in these examples. The job descriptor for the *reader_writer* object is defined by the **interface** section in figure 4.2. In the *reader_writer*

object, *job(i).service* references the service tag field. Boolean guards may also test the value of the mediator's local variables. Boolean guards paired with status tests are not evaluated if the status test is false.

In the following explanation of a guarded command, the execution of the guard is considered in isolation, without considering possible interleaving with other parallel guarded commands. The presence of parallel guards introduces delays, but does not affect the semantics of the guarded command.

Mediator guarded commands are closely related to Brinch Hansen's guarded regions [18]. The mediator process must wait until some guard condition is true, and then execute the associated statement list. A statement list associated with a *true* guard is said to be *enabled*. A guard whose associated statement list has been chosen and started execution is said to have been *fired*.

When the statement list of a fired guard has finished executing, the exit condition in the final **until** line of the guarded command is tested. If the condition is true, the guarded command terminates, otherwise its guards are reevaluated.

Nondeterminism is a possibility when more than one guard is enabled. In this case, one guard will be chosen to fire. A mediator implementation must ensure at least weak fairness to avoid starvation problems. Weak fairness in guard evaluation means that a guard that is enabled often enough will eventually be fired. The mediator cannot delay if there are enabled guards.

The *delay* semantics of this guard command differs from Dijkstra's original definition and Hoare's adaptation [32,50]. Hoare and Dijkstra's constructs abort the guarded command

when no guard is true. This creates a framework that is convenient for formal verification, but results in servers that do not facilitate waiting. Waiting is usually implemented by explicitly programming a busy loop. Because waiting is fundamental to providing services, we prefer to wait implicitly.

Brinch Hansen implements both *delay semantics* in guarded regions and *abort semantics* for guarded commands. The mediator proposal includes only *delay semantics*, because the inclusion of an **otherwise** guard and exit conditions make the abort semantics redundant. The **otherwise** guard has other applications for implementing background actions and is a useful shorthand for the negation of all other guards.

Mutual exclusion within a mediator depends both on the use of the **exec** statement and the careful choice of preconditions defined in a guard statement. The **exec** statement initiates a service process and blocks the mediator, but it does not check for other initiated processes. In the reader–writer example (figure 4.2), mutual exclusion for the *write* operation is ensured by the guard "**cycle** *reader_count* = *0* –>" and by the action of **exec**. The guard will not permit a *write* to begin until all executing *read* operations are terminated. The **exec** statement blocks the mediator as the *write* is serviced to prevent other operations from becoming active.

*Keys* are used to identify the client to the mediator, to access job descriptors for guard evaluation and scheduling purposes and to tie clients to specific resources, as in allocator objects. The *key* concept was suggested by Hoare's CSP process range labels [50,52], but their use in mediators is considerably different. Hoare applies ranges to processes to create a finite number of explicitly and contiguously indexed processes. This application of ranges is not included in mediators. Hoare also applies ranges to guarded commands to substitute

values within a given range for a bound variable in the guard statements. The following example is from [50]:

> "(*i*:1..*n*)G -> CL stands for
> G1 -> CL1 □ G2 -> CL2 □ ... □ Gn-> CLn."

In effect, the guard is expanded by creating a guard and statement list for every value of *i*. The application of ranges in Hoare's guarded commands is quite general.

In the mediator proposal, keys serve only to identify client processes. Like Hoare's ranges, a key statement (**any** ...) defines a key variable which will be bound within the guard command it modifies. Consider the guarded command shown in figure 4.6. It is executed as if it were written as shown in figure 4.7. In this example the value of the key identifier is in the range *1..n* and defined as the interface field *job.rangeprm*. Usually a process identifier (the *pid* descriptor field) will be used as the key. The designer of a mediator does not need to know explicitly what process identifier values are being used, just that they are unique. Although, in an abstract sense, a potentially infinite key variable range implies an infinitely expanded guard, there is no need to implement them that way. Keys are always associated with status tests. Only guards corresponding to clients with requests can evaluate to *true*, so

```
any i in key: cycle
                req(i); job(i).service = A ->
                        exec(i,A);
                        release(i);
                □
                req(i); job(i).service = B ->
                        x := x + 1;
                        exec(i,B);
                        release(i)
        until false;
```

Figure 4.6. A Guarded Command.

```
cycle
        req(1); job(1).service = A ->
                exec(1,A);
                release(1);
        □
        req(1); job(1).service = B ->
                x := x + 1;
                exec(1,B);
                release(1);
        □
        req(2); job(2).service = A ->
                exec(2,A);
        ...

        □
        req(10); job(10). service = B ->
                x := x + 1;
                exec(10,B);
                release(10)
until false;
```

Figure 4.7. The Guarded Command Expanded.

only such guards need to be evaluated. This significantly limits the number of guards evaluated. Evaluation can be restricted further when fairness is taken into consideration.

Key variables are tied to job descriptors defined by the interface. The most useful key reference is to the client process identifier. The mediator designer may designate another descriptor field as a key, as in figures 4.6 and 4.8. In any case, the chosen key field must be unique for each pending request.

The mediator in figure 4.8 implements synchronization for the dining philosophers problem. The client process executes the statement *diner (rangeprm, eat);* to request the mediator's *eat* service. This solution is one of many possible solutions using mediators.

```
diner = object
  interface
    job: export part
      myfork : key range;
      case service: (eat) of
          eat: ()
    end export part;

  procedure eat;
  begin    - do whatever to eat   end;

  mediator
    type  range = 0 .. n-1
    var
          fork : array [range] of (free, inuse);
          i, j   : range;
    init
      for j := 0 to n-1;   fork[j] := free;
    end init
    body
      any i in key: cycle
            req(i); job(i).service = eat and fork[i] = free
                  and fork[(i+1) mod n] = free ->
              fork[i] := inuse;
              fork[(i+1) mod n] := inuse;
              spawn(i, eat(i));
            □
            term(i); job(i).service = eat ->
              fork[i] := free;
              fork[(i+1) mod n] := free;
              release(i);
        end cycle
    end body
  end mediator
end object
```

Figure 4.8. Dining Philosophers.

## 4.5. Parallel Guarded Commands

The schematic in figure 4.9 shows the syntax of the parallel guarded command, a mechanism that allows the interleaving of different mediator actions.

```
body
        guarded_command
        //
        ...
        //
        guarded_command
end body
```

Figure 4.9. Parallel Guarded Command Schematic.

Parallel guarded commands are proposed to allow different sets of guards to be evaluated at different times during mediator execution. It allows the mediator to "shuffle" together the evaluation of several guarded commands. The choice of the notation // to separate parallel guarded commands is deliberate. A mediator containing parallel guarded commands uses a multiprogrammed thread of control, one thread of control for each guarded command. Only one thread of control is active at a time. The active control block can change only when guards are evaluated. This creates mutually exclusive execution of the statement lists between guard evaluations. The mediator body terminates if all of the parallel guard blocks terminate.

Consider the simplified example in figure 4.10. (Labels have been included to make discussion easier). In figure 4.10, *A, B, C, D* are guards. *SA, SB, SC, SD* are statement lists. The control vector of this mediator has two elements. The notation "$<$*label1, label2, ... , labeln* $>$" is a control vector in which n threads of control are at the locations label1 through labeln. This notation is adapted from the expression of execution state in Manna and Pnueli's temporal logic scheme[84]. In figure 4.10, the initial control vector is: <l1, m1>. When guard evaluation occurs in the initial state, the guards *A* and *C* are evaluated. As for isolated guard commands, the associated statement list of some true guard will be executed.

```
body
  l1: cycle
        A ->   l2: SA;
               l3: cycle  B ->   l4: SB  until true;
    until false
      //
  m1: cycle
        C ->   m2: SC;
               m3: cycle  D ->   m4: SD  until true;
    until false
end body
```

Figure 4.10. Simplified Parallel Guarded Command.

If the guard *A* from the cycle l1 is fired, the statement list starting at l2 will begin execution. It will continue executing without interruption until the new guard command at l3 is encountered (assuming SA contains no guard commands). At this point the control vector is <l3, m1>, and the new guard evaluation includes the guards *B* and *C*. Considering all possible combinations, the set of guards evaluated at any one time may be: [A, C], [A, D], [B, C] or [B, D].

The statement lists following guards may contain **exec**, **spawn** and **release** statements without altering the flow of control discussed above. In every case, control passes to the following statement. In the case of an **exec** statement, this is delayed until the resource operation it has initiated terminates. This delay temporarily blocks further mediator activity, but does not alter the flow of control.

The parallel guard notation is an easy and concise way of specifying changing sets of enabling conditions. It is possible to rewrite a parallel guard as one large simple guard command by using a distribution algorithm. The resulting guard command is considerably more bulky and actually less clear.

The introduction of a control vector within the mediator does not create the same complications for reasoning about programs that are usually associated with parallel processes. The control flow in mediators is very restricted, giving statement lists that will be executed in mutual exclusion. This fact, combined with the small size of mediators and the explicit statement of preconditions in the guards makes it quite easy to reason about the behavior of parallel guards.

The reader/writer mediator demonstrates one application of the parallel guard. In that example, firing the guard *req(i); job(i).service = write* executes the associated statement, which is a *cycle* statement. As long as its guard *reader_count = 0* is false, the guard cannot fire. No new *write* or *read* operations will be initiated, but the second parallel guard will allow *read* operations to finish up and leave the mediator. Parallel guarded commands coupled with nested guard commands gives a convenient way to block some actions while permitting others.

### 4.6. Some Additional Examples

The examples that follow demonstrate some applications of mediators. In many cases only the mediator will be presented.

#### 4.6.1. Alarm clock

The alarm clock object (figure 4.11) delays a caller for a time period specified in the request's parameter *n*. Calls for the *wake* service cause a delay. Calls for the *tick* service advance the clock. The field *out_time* must be declared for the operation *wake* job descriptor within the mediator as a mediator local extension to the job descriptor. This figure presents

```
alarm_clock = object;
  interface
    job : export part
      pid : key client_process_id;
      case service : (wake, tick) of
        wake : (n : integer);
        tick : ()
      end case
    end export part

  procedure wake; begin  end;
  procedure tick; begin  end;

  mediator
    var
      i, j : client_process_id;
      now  : integer;
      flag : boolean;

      job  : local part
        case service : (wake, tick) of
          wake : (out_time : integer);
          tick : ()
        end case;
      end local part

    init now := 0 end init
    body
      any i in key: cycle
        req(i); job(i).service = wake ->
              -- start the service, but termination will be delayed
              job(i). out_time := now + job(i). n;
              spawn(i, wake);
      until false;
      //
      any i in key: cycle
        req(i); job(i).service = tick ->
              now := now + 1;
              exec(i, tick);
              release(i);
              flag := false;
              any j in key: cycle
                      term(j);  job(j).service = wake and job(j).out_time <= now  ->
                        release(j);
                      □
                      otherwise -> flag := true
                        -- exit cycle
                  until flag;
      until false;
    end body
  end mediator
end object
```

Figure 4.11. Alarm Clock.

a complete mediated object in order to demonstrate the use of a job descriptor augmented by local data.

### 4.8.2. Shortest Job Next

The mediator in figure 4.12 implements a scheduler that chooses the job with the lowest estimated service time for the next execution. Requests are served in mutual exclusion. This framework is applicable to many scheduling problems.

The first guard command simply calls a local operation to queue up job descriptors in order of their estimate parameter. The second guard command removes the head element of the job descriptor queue and starts that job's execution. The *spawn* and wait for termination allows the mediator to continue enqueueing new requests while a service operation is executing.

```
body
  any i in key: cycle
          req(i); job(i).service = server ->
               enqueue (i, job(i).estimate);
       until false;
       //
   cycle
       queue_not_empty ->
          j := dequeue;
          spawn(j, server);              -- initiate service operation
          cycle
               term(j); job(j).service = server ->
                  release(j);
          until true;
   until false;
end body
```

Figure 4.12. Shortest Job Next.

The key variable *j* in the second guard command is set by direct assignment rather than through a **cycle** modifier.

### 4.6.3. An Allocator

The allocator in figure 4.13 gives a client process exclusive rights to a resource for a series of accesses. The client must request an allocation, then may make repeated calls on the resource. Finally, the client must explicitly release the resource before it can become available to another client. This example uses the key binding made in the outer **cycle** to restrict use of the resource to one process in the inner **cycle**.

The mediator differs from the monitor solution [49] for this problem in a number of ways. Most importantly, the resource being allocated is encapsulated with the mediated object. The mediator protects the resource from unsynchronized accesses by faulty processes.

```
body
  any i in key: cycle
          req(i); job(i). service = allocate ->
                exec(i, allocate);
                release(i);
                flag := false;
                cycle
                   req(i); job(i).service = use ->
                         exec(i, use);
                         release(i);
                   □
                   req(i); job(i).service = free ->
                         exec(i, free);
                         release(i);
                         flag := true;
                until flag;
        until false;
end body
```

Figure 4.13. Allocator.

```
init  val := 0  end init
      -- val is the semaphor counter variable
body
   any i in key: cycle
           req(i); job(i). service = P ->
                 exec(i, P);
                 val := val + 1;
                 release(i);
         □
           req(i); job(i). service = V and val > 0 ->
                 exec(i, P);
                 val := val - 1;
                 release(i);
         until false;
end body
```

Figure 4.14. Semaphore.

The mediator also prevents the resource from being released by any process but the one the resource has been granted to. The monitor solution does not offer protection in either of these cases.

### 4.6.4. A Semaphore

The mediator in figure 4.14 implements a semaphore. We have included this example to demonstrate that the mediator construct has at least the expressive power of semaphores. The function of this object is purely synchronization.

### 4.6.5. Cigarette Smokers Problem

Patil[96] introduced this problem to demonstrate some of the limitations of semaphores for implementing synchronization:

> Three smokers are sitting at a table. One of them has tobacco, another has cigarette papers, and the third has matches; each one has a different ingredient required to make and smoke a

> cigarette but he may not give an ingredient to another. On the table in front of them, two of the three ingredients will be placed, and the smoker who has the necessary third ingredient should pick the ingredients from the table, make a cigarette and smoke it.

A mediated object solution to this problem is presented in figure 4.15. In this example, three processes (*deliver_paper, deliver_tobacco, deliver_match*) place items on the table. The three smoker processes are got_match, got_paper and got_tobacco.

### 4.6.6. Ring Buffer

Ring buffers are frequently used in operating systems to support processes that act in a producer/ consumer relationship[97]. The producers fill in the buffers at their own pace, while consumers empty those buffers. In a ring buffer, the number of available buffers is limited and the total number of buffers is fixed. Synchronization is used both to prevent interfering accesses to a single buffer and to prevent consumers from "overtaking" the producers. A simple solution to the problem that allows at most one simultaneous *put* and *get* on different buffer is presented in figure 4.16. We have include the entire mediated object.

```
init
   tobacco := false;  paper   := false; match   := false;
   resource_count := 0;
end init
body
   cycle
      resource_count < 2 ->
         any i in key: cycle
            req(i); job(i). service = deliver_tobacco and not tobacco ->
               exec(i, deliver_tobacco);
               tobacco := true;
               resource_count := resource_count + 1;
               release(i);
            □
            req(i); job(i). service = deliver_paper and not paper ->
               exec(i, deliver_paper);
               paper := true;
               resource_count := resource_count + 1;
               release(i);
            □
            req(i); job(i). service = deliver_match and not match ->
               exec(i, deliver_match);
               match := true;
               resource_count := resource_count + 1;
               release(i);
         until true;
      □
      resource_count = 2 ->
         any i in key: cycle
            req(i); job(i). service = got_match and tobacco and paper ->
               exec(i, got_match);
               tobacco := false; paper  := false;
               resource_count := 0;
               release(i);
            □
            req(i); job(i). service = got_paper and tobacco and match ->
               exec(i, got_paper);
               tobacco := false; match  := false;
               resource_count := 0;
               release(i);
            □
            req(i); job(i). service = got_tobacco and paper and match ->
               exec(i, got_tobacco);
               paper := false; match  := false;
               resource_count := 0;
               release(i);
         until true;
   until false;
end body
```

Figure 4.15. Cigarette Smokers.

```
ring_buffer = object;
  interface
    job : export part
      pid : key client_process_id;
      case service : (put, get) of
        put : (p_data : some_type );
        get : (g_data : var some_type)
      end case
    end export part

  const ring_size = n;
  type  ring_range = 0 .. (n-1);
  var   ring : array [ring_range] of some_type;

  procedure put (data: some_type, ring_index : ring_range);
  begin ring [ring_index] := data end;

  procedure get (data: var some_type; ring_index: ring_range);
  begin data := ring [ring_index] end;

  mediator
    var
      i : client_process_id;
      head, tail : ring_range;

    init head := 0; tail := 0 end init
    body
      any i in key: cycle
        req(i); job(i).service = put and (head + 1) mod ring_size < > tail ->
            spawn(i, put (job(i). p_data, head));
            cycle
              term (i); job(i).service = put ->
                release (i);
                head := (head + 1) mod ring_size;
            until true;
      until false;
      //
      any i in key: cycle
          req(i); job(i).service = get and head < > tail ->
            spawn(i, get (job(i). g_data, tail));
            cycle
              term(i); job(i). service = get ->
                release (i);
                tail := (tail + 1) mod ring_size
            until true;
       until false;
    end body
  end mediator
end object
```

Figure 4.16. Ring Buffer.

# CHAPTER 5.

# A TEMPORAL LOGIC SPECIFICATION

In the chapter above we presented an informal description of the mediated object. Such a description is useful as an introduction to a new language construct, but is not adequate as a basis for implementation or for gaining a detailed knowledge of the construct. In this chapter we present a temporal logic specification of the mediated object construct. Parts of this specification were originally developed in [39].

The imprecision of informal specifications led to a search for a practical means of formally specifying the mediator construct. The specification tool chosen had to meet these criteria: naturally describe concurrency; easily deal with shared variables; be independent of possible implementation; serve as a basis for verification. Temporal logic [84] meets these requirements. Because temporal logic makes synchronization relationships explicit, it is a powerful tool for reasoning about concurrent programs. This chapter presents a formal temporal logic specification of mediators which can serve as the outline of a proof system to support verification.

The use of temporal logic to specify a new programming construct is a new and, we believe, powerful technique. Research in temporal logic has primarily applied this tool to demonstrate verification techniques using simplified languages [70,84,94]. It also has been used, as a secondary specification for program synthesis or verification purposes, for languages that have already been formally specified by other means [98,121].

We had hoped to be able to directly apply Manna and Pnueli's model [81] to mediators, but were soon disabused of that notion. Both the data domain and control structures used in mediators are more complex than those of Manna and Pnueli's simplified language. Our constructs are also based on somewhat different assumptions. Much of our work has involved developing a different model for mediators and adapting temporal logic axiomatization to this model.

### 5.1. Temporal logic as a language specification tool

We considered several language specification methods before temporal logic was chosen to specify mediators. These included attribute grammars, axiomatic semantics and denotational semantics.

An attribute grammar definition was rejected for a number of reasons. Attribute grammar definitions give an operational definition of a language that can be very suggestive of an implementation. For this reason they tend to be very useful to implementers, but unhelpful to users [95]. A language specified using attribute grammars usually would need another specification to support verification. Finally, attribute grammars tend to be very large with a lot of their bulk devoted to elaborating data typing and basic language elements. Because the mediator construct is meant to be an extension to an unspecified Pascal-like language, a specification technology that would allow us to be somewhat unspecific about data typing and concentrate on concurrency issues is preferable.

Denotational semantics are abstract and precise enough for our use, but the application of denotational semantics to concurrency problems is very much a research topic [110]. Moreover, denotational descriptions do not deal well with shared variables. Finally, denota-

tional descriptions are extremely complex and difficult to understand. For these reasons this approach was rejected.

Axiomatic specifications offered a tool that allowed sufficient abstraction and precision without undue complexity. They promised to be useful and meaningful to a wider range of users and implementers than either attribute grammars or denotational semantics would be. Although a great deal of research has been done on applying axiomatic semantics to concurrent programming [53,69,92,93], axiomatic definitions do not appear to be well adapted to expressing complex interactions between multiple flows of control.

Temporal logic builds on an axiomatic basis by adding a direct and natural means of reasoning about the sequences of events and flow of control in a concurrent system [70,84,94]. It can be used as a tool to provide an unambiguous description of mediator behavior for a potential implementer, and a sound basis for verification for a user. It already has provided a check on the mediator design specification during the development of the mediator construct. Temporal logic has proved to be an excellent tool for the specification of mediators.

## 5.2. A Short Introduction to Temporal Logic

In this section we present a short and informal introduction to temporal logic. The subject is explored in great detail in[81,82,84], and interested readers may look there. Our discussion follow the outline of the introduction presented in[64].

Temporal logic is a first–order language that uses the familiar logical operators, quantifiers and connectors. In addition four *modal* operators are defined:

- $\Box$ the "always" operator,
- $\Diamond$ the "sometimes" operator,
- $\circ$ the "next" operator,
- $\mathcal{U}$ the "until" operator.

The first three operators are unary, the last is binary.

Manna and Pnueli partition variables into global and local sets. The global variables are unchanged over the execution of the program, while local ones may change from step to step in the computation. Quantifiers are only applied to the global variables.

Manna and Pnueli's temporal logic language uses a model(I,$\alpha$,$\sigma$) in which I is a global interpretation, $\alpha$ a global assignment and $\sigma$ a sequence of states. I specifies the domains in which the language operates. $\alpha$ defines the value of all global variables. $\sigma$ is the component of the language that is, in a sense, of the most interest to us. Temporal formula are defined over infinite sequences of states $\sigma$:

$$\sigma = s_0, s_1, s_2, \ldots$$

Each state $s_i$ gives the value of all the variables. The global variables do not change value from state to state. The local variables may change value.

It is also possible to speak of sequences of states that do not originate with the initial state $s_0$. These can be referred to as k–shifted states denoted:

$$\sigma^{(k)} : s_k, s_{k+1}, \ldots$$

The temporal operators can be interpreted over sequences of states as follows. The notation "s $\vdash$ w" means that formula w is interpreted over the state sequence $\sigma$.

Since logical expressions that do not contain temporal operators are time independent, they can be expressed as:

$\sigma \vdash$ w iff $s_0 \vdash$ w

The unary temporal operators can be written as:

$\sigma \vdash \Box$ w iff $\forall$k : k$\geq$ 0: $\sigma^{(k)} \vdash$ w
$\sigma \vdash \Diamond$ w iff $\exists$k : k$\geq$ 0: $\sigma^{(k)} \vdash$ w
$\sigma \vdash \circ$ w iff $\sigma^{(1)} \vdash$ w

The "until" operator can be defined (using a quantifier notation developed in[41]. ):

$\sigma \vdash w_1 \; \mathcal{U} \; w_2$ iff $\exists$k :k$\geq$ 0: $\sigma^{(k)} \vdash w_2$ and $\forall$i : $0 \leq$ i $\leq$ k: $\sigma^{(i)} \vdash w_1$

Manna and Pnueli define other operators and give detailed definitions of other logical operators and quantifiers in[83]. In our notation we represent a vector y by a y with a bar over it: $\bar{y}$.

A program can not be verified using temporal logic without a temporal proof system the programming language in which the program is written. The proof system consists of three parts[84]. The first of these is the *uninterpreted logic part*, which essentially defines the axioms of first–order temporal logic. This portion does not change for any programming language, and is defined in[83,84].

The second part of the temporal proof system is the *domain part* which defines the domains of a programming language and any induction rules covering those domains. In the case of mediators, a great deal of the domain definition pertains to a potential host language, and not to the mediated object. As a result, to simplify matters we have chosen to deal only with natural numbers within our mediated object language. These can be dealt with using familiar axioms.

One useful example of an induction rule pertains to to sets with a *well founded* ordering. A set A is said to be *well founded* with respect to an ordering relation $\succ$ if there exists no

infinitely decreasing sequence

$$\alpha_0 \succ \alpha_1 \succ \alpha_2 \succ \cdots$$

If (A,$\succ$) is a well founded set and w($\alpha$) is a temporal formula dependent on $\alpha \in$ A, then an induction rule can be defined that will allow us to prove the termination of an iterative language construct. One such rule is the $\diamond$IND rule, which can be stated as:

$$\frac{w(\alpha) \supset \diamond[\psi \vee \exists\beta : \beta \succ \alpha : w(\beta)]}{w(\alpha) \supset \diamond\psi}$$

This rule serves as a kind of a template which can be filled in to "instantiate" rules for various kinds of well founded sets. For example, an instantiation of this rule for natural numbers called "IND" is given in [80].

$$\frac{Q(0) \supset \diamond\psi \qquad Q(m+1) \supset \diamond \vee \diamond Q(m)}{Q(k) \supset \diamond\psi}$$

If Q is a predicate associated with the loop, and $\psi$ is the loop termination condition, the this rule can be used to show the termination of a loop.

The third part of a temporal proof system is the *program part* in which the semantics of the programming language are defined. Manna and Pnueli define their languages using a set of graph templates[84]. We will follow their method in this introduction. In our temporal logic definition of the mediated object we chose to use a textual method that is equivalent.

In a temporal proof system, a program may be represented as a directed graph. Variable declarations are not included. The graph abstracts the program's flow of control. For example, a Pascal **if** statement would be represented by two edges coming out of a common vertex. One of these would represent the **then** branch of the **if**; the other would represent

the **else** branch. Both of these edges join again at the vertex representing the exit of the if statement. The edges of the graph are each labeled and tagged by a guarded command. The guard states the conditions under which that particular transition may be taken. The statement part of the guarded command may either be null or an assignment. These are the only statments left after the control flow of the program has been reduced to a graph. Figure 5.1 shows an **if** statement and its graph. The edges a and b in figure 5.1 represent the **then** and **else** part of the **if** statement, respectively. Both have a source vertex at l and a sink vertex at m. This approach tends to strip away most of the syntactic aspects of the language leaving only the semantic core of the language behind.

In Manna and Pnueli's model of a concurrent program $P$ with $m$ parallel processes

$$\text{P: } \bar{y} := g(\bar{x}); P_1 \parallel \ldots \parallel P_m$$

is represented in this system by a number of these graphs, one for each process in the program[83,84]. Each process graph has a unique initial vertex. A concurrent process graph may or may not have a termination vertex, reflecting the fact that many concurrent

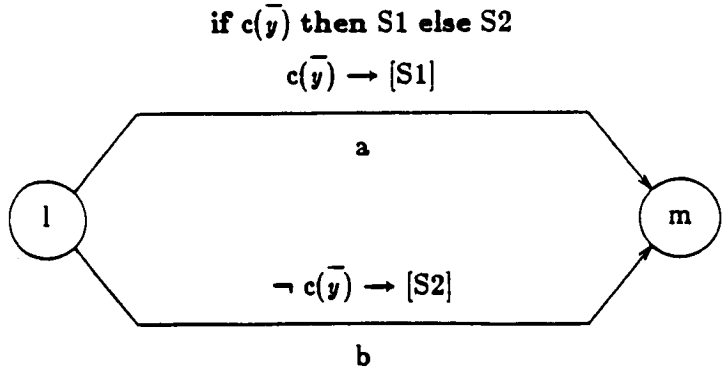



Figure 5.1. if graph.

programs have continuously operating processes.

A state of a concurrent program is of the form:

$$s = \langle \bar{\lambda} ; \bar{\eta} \rangle$$

where $\bar{\lambda} = \langle \lambda_1, \ldots, \lambda_m \rangle$ is a vector of the current values held by the location counters of each process $\pi^i$ and $\bar{\eta}$ is a vector of the current values of all the local program variables $\bar{y}$. Each element in $\bar{\lambda}$ points to the next instruction to be executed in its process[83]. Program execution is modeled by an infinite sequence of states $\sigma$, as for sequential programs.

Proofs of simple sequential programs in a temporal proof system proceed by assuming that initially program control is at the initial vertex (a vertex of in-degree 0) and then formally showing that eventually the last vertex must be reached with a certain predicate holding. Sequential program proofs are either partial correctness proofs or total correctness proofs. In a partial correctness proof, it must be demonstrated that if a program starts at the initial vertex and a correct initial state, that if the last vertex is reached a certain predicate will hold. Partial correctness proofs do not require a proof that the last vertex will ever be reached, just that a certain condition will be true *if* that occurs. In a total correctness proof, termination must be proved as well[41,45].

These kinds of proof may not apply in concurrent programs because frequently concurrent programs are continous and contain no terminal vertices. Instead concurrent proofs are concerned with *safety (invariance)* and *liveness (eventuality)* properties[68,82,84,94].

Safety properties describe what states are permissible during concurrent program execution. In this way they insure that "nothing bad will ever happen"[68]. Partial correctness is one kind of safety property. Others include mutual exclusion and freedom from deadlock[84].

These properties may be specified by a predicate that must hold for each transition in a program. For this reason they are often called invariance properties.

Liveness properties state that certain things *must* occur[68]. That is, that once a certain state has been reached in a computation, that eventually some other state will hold. Total correctness is one example of a liveness property in that once the initial state holds, total correctness demands that the termination vertex must eventually be reached and a certain predicate must hold. Other liveness properties of concurrent programs include the accessiblity of critical sections, responsiveness and liveness[84].

The notion of being "at a vertex" may be made more formal by introducing *location variables*. A location variable can point to a location on a graph (or in a program) and allows a concise expression of where the control is in a given program. For example, the expression "at l" says that control is at the first vertex of the if statement in figure 5.1. Location variables can further be used to express control flow with direct reference to program text rather than graphs, as we have done below.

Once a program graph has been drawn, axioms can be written to express the effect of each edge (or transition) in the graph. These axioms are called transition axioms and reflect exactly the effect of taking a transition in the graph. In this way, they provide a formal specification of the programming language.

For example, the Pascal if statement has the transition axioms:

$$F_a\text{: } [\text{ at l} \wedge \text{c}(\bar{y}) \wedge \bar{y}{=}\bar{u}] \supset \Diamond[\text{ at m} \wedge \bar{y}{=}\overline{F(u)}]$$

$$F_b\text{: } [\text{ at l} \wedge \neg\text{c}(\bar{y}) \wedge \bar{y}{=}\bar{u}] \supset \Diamond[\text{ at m} \wedge \bar{y}{=}\overline{G(u)}]$$

(assuming that S1 is $\bar{y} := \text{F}(\bar{y})$ and S2 is $\bar{y} := \text{G}(\bar{y})$)

Each transition axiom has a name that relates it to an edge on the graph to which it refers. Transition axioms state that if control is at a certain vertex of the graph, and certain conditions hold, that eventually control will be at another vertex with another set of conditions holding.

In the example above, axiom $F_a$ states that if control is at label l and c($\bar{y}$) holds and $\bar{y}$ is equal to some set of global variables $\bar{u}$, then sometime control will pass to label m and $\bar{y}$ will have been changed according to function F. The meaning of axiom $F_b$ is similar.

The use of global variables in the transition axioms (such as $\bar{u}$ in the axioms above) is a standard trick that makes it easier to deal with the fact that local variables change during state transitions. Because of this change over time, it is not meaningful to speak in terms of $\bar{y} := f(\bar{y})$. In a temporal proof system, for a given state the $\bar{y}$ on the one side of the assignment must have the same value as the $\bar{y}$ on the other side. An assignment of this sort only makes sense if f is an identity function. Global variables are used to "freeze" the values of $\bar{y}$ in one state to make the assignment meaningful.

In general, both in this short introduction and in the temporal logic specification for mediators that follows, all local program variable names will start with a "y". Global variables will start with an "x". Auxiliary global variables (as in the axioms above) will start with a "u".

The axioms presented in this introduction are so–called *weak transition axioms*[80] because they use the sometimes operator $\Diamond$. For sequential programs, stronger axioms could be written using the next operator $\circ$.

All edges in a graph, or statements in a language, can be formally axiomatized in this manner. Figure 5.2 shows a generalized graph of a statement transition.

The program part of a temporal proof system is used to define the semantics of a particular language. This usually includes giving templates for constructing graphs from the language's control abstractions. It also includes defining the semantics of a statement that could label the command part of the guarded commands on the graph edges. An example of a complete language specified this way can be seen in[63]. This thesis presents a specification of a portion of a language using textual representations in place of graphs.

Once such a specification has been completed, it can be used as part of a temporal proof system to verify programs. In general, to prove a safety property of a concurrent program of the form

$$\mathrm{P}: \bar{y} := \mathrm{g}(\bar{x});\ P_1 \parallel \ldots \parallel P_m$$

we show that:

$$\varphi(\bar{x}) \supset \Box\, \psi$$

where $\varphi$ is the precondition of the program and $\psi$ an invariant. In order to prove this invariant, it is necessary to show that

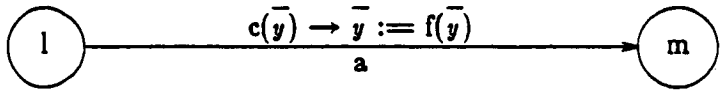



Figure 5.2. Generalized graph.

- the assertion $\psi$ is true in the initial state of the program:
  e.g.: $\varphi(\overline{ymed}) \supset \psi$

- the assertion $\psi$ remains true for all possible transitions.
  e.g.: for every transition in P, if the assertion $\psi$
  is true before the transition, it remains true afterwards.

This proof rule is formalized in [83] Where it is called the *Initialized Invariance Rule* (IINV):

$[at\ \bar{l}_0 \wedge \bar{y} = g(\bar{x})] \supset \psi$
P leads from $\psi$ to $\psi$

---

$\Box\, \psi$

Liveness properties may be proved using the *Eventuality Rule* (EVNT) formailized in[83]. In this rule $\varphi$ and $\psi$ are again functions of the program state ($\varphi(\bar{\pi};\bar{y})$; $\psi(\bar{\pi};\bar{y})$) and $P_k$ is one process of program P:

A: P leads from $\varphi$ to $\varphi \vee \psi$
B: $P_k$ leads from $\varphi$ to $\psi$
C: $\varphi \supset \Diamond(\psi \vee Enabled(P_k))$

---

$\varphi \supset \varphi \mathcal{U} \psi$

The propositions A,B and C that allow us to establish the conclusion are fairly straightforward. Proposition A states that the transitions of $P$ either must maintain the truth of $\varphi$ or establish $\psi$. Proposition B requires us to show that a transition by one process $P_k$ goes from a state satisfying $\varphi$ to one satisfying $\psi$. The final proposition C requires us to show that if we are in a state satisfying $\varphi$, that eventually either $\psi$ will be true, or a transition of $P_k$ will be enabled.

The EVNT rule also can be used to establish liveness properties of the form:

$$\varphi \supset \Diamond \psi.$$

These two proof rules (IINV and EVNT) are essential to the proof of the sample concurrent program that calculates the Greatest Common Denominator (GCD) that is presented in[83] (figure 5.3). The sample proof that follows is taken from that paper and has been altered only slightly to add additional explanation where needed.

The GCD program is meant to terminate with the correct greatest common denominator in variable $y_1$. It requires a total correctness proof of the theorem:

$$[\text{at } (l_0,m_0) \wedge (y_1,y_2) = (x_1,x_2)] \supset \Diamond[\text{at } (l_2,m_2) \wedge y_1 = \gcd(x_1,x_2)]$$

The proof of theorem a can be split into a proof of a safety (invariance) property and a proof of termination (a liveness property).

The invariant we must prove is:

$$\text{(Lemma A) } \Box[\gcd(y_1,y_2) = \gcd(x_1,x_2)]$$

To prove lemma A:

A1. Show $\gcd(y_1,y_2) = \gcd(x_1,x_2)$ initially,
A2. Show $\gcd(y_1,y_2) = \gcd(x_1,x_2)$ remains true over all transitions in P.

Since initially

---

$$(y_1, y_2) := (x_1, x_2)$$

| $-P_1-$ | $-P_2-$ |
|---|---|
| $l_0$: **if** $y_1 > y_2$ **then** $y_1 := y_1 - y_2$ | $m_0$: **if** $y_1 < y_2$ **then** $y_2 := y_2 - y_1$ |
| $l_1$: **if** $y_1 \neq y_2$ **then go to** $l_0$ | $m_1$: **if** $y_1 \neq y_2$ **then go to** $m_0$ |
| $l_2$: **halt** | $m_2$: **halt** |

Figure 5.3. Distributed Greatest Common Denominator

---

$$(y_1y_2) = (x_1,x_2),$$

the first proposition is obviously true. It is easy to show that the second proposition holds over all the transitions of P as well, so we deduce that the invariant in lemma A holds.

Manna and Pnueli prove a second lemma (lemma B) in[83] to help prove termination:

Lemma B:

$$[\text{at } l_{0,1} \wedge \text{at } m_{0,1} \wedge (y_1,y_2) > 0 \wedge (y_1,y_2) \leq \text{n+1} \wedge y_1 \neq y_2] \supset \Diamond[\text{at } l_{0,1} \wedge \text{at } m_{0,1} \wedge (y_1,y_2) > 0 \wedge (y_1,y_2) \leq \text{n})]$$

The notation *at* $l_{0,1}$ is used as an abbreviation for *at* $l_0 \vee$ *at* $l_1$, and the notation $(y_1,y_2) > 0$ means $(y_1) > 0 \wedge (fIy_2) > 0$.

Proof of lemma B:

First, it is useful to define a predicate:

$$\varphi(y_1,y_2, \text{n}): \text{at } l_{0,1} \wedge \text{at } m_{0,1} \wedge (y_1,y_2) > 0 \wedge (y_1 + y_2 \leq \text{n}).$$

So lemma B is:

$$[\varphi(y_1,y_2,\text{n+1}) \wedge (y_1 \neq y_2)] \supset \Diamond\varphi(y_1,y_2,\text{n}).$$

The proof can be split into two cases:

$$\text{B1. } [\varphi(y_1,y_2,\text{n+1}) \wedge (y_1 > y_2)] \supset \Diamond\varphi(y_1,y_2,\text{n}).$$
$$\text{B2. } [\varphi(y_1,y_2,\text{n+1}) \wedge (y_1 < y_2)] \supset \Diamond\varphi(y_1,y_2,\text{n}).$$

To prove B1 Manna and Pnueli [83] note that by the rules of propositional reasoning:

$$1.\ \varphi(y_1,y_2,\text{n+1}) \supset (\text{at } l_0 \vee \text{at } l_1).$$

The first case to consider is when process $P_1$ is *at* $l_0$. Let:

$\varphi'$: $\varphi(y_1,y_2,\text{n+1}) \wedge (y_1 > y_2) \wedge$ at $l_0$
$\psi'$: $\varphi(y_1,y_2,\text{n})$

Propositions $\varphi'$ and $\psi'$ satisfy the premise of the EVNT rule with $P_k = P_1$.

First consider proposition A of the EVNT rule. This states that every transition in P must lead from $\varphi'$ to $\varphi' \vee \psi'$. If we look at the transitions of $P_2$, we notice that only the transition $m_0 \rightarrow m_1$ and the transitions from $m_1$ are relevant. If $y_1 > y_2$ when the transition $m_0 \rightarrow m_1$ occurs, the value of $\varphi'$ does not change. Since only the transitions out of $m_1$ to $m_0$ is possible if $fIy_1 > fIy_2$, and this transition does not affect the local variables, they also leave $\varphi'$ invariant. This establishes proposition A of EVNT.

In the process $P_1$, only the transition $l_0 \rightarrow l_1$ is enabled. The effect of this transition is to replace $(y_1,y_2)$ by $(y_1 - y_2, y_2)$. If before this transition $y_1 + y_2 \leq n + 1$ and $(y_1,y_2) > 0$, then by simple arithmetic, $\psi'$ will hold after the transition. This establishes proposition B of EVNT with $P_k = P_1$.

Since by $\varphi'$, the transition from $l_0$ is enabled, proposition C is immediately true. Since we have demonstrated all three conditions of the EVNT rule we may conclude that $\varphi' \supset \Diamond\psi'$, that is:

$$2.\ [\varphi(y_1,y_2,n+1) \wedge (y_1 > y_2) \wedge \text{at } l_0] \supset \Diamond\varphi(y_1,y_2,n).$$

The next subcase of B1 is where $P_1$ is *at* $l_1$. Again we define two predicates:

$\varphi''$: $\varphi(y_1,y_2,n+1) \wedge (y_1 > y_2) \wedge \text{at } l_1$
$\psi'' = \varphi'$: $\varphi(y_1,y_2,n+1) \wedge (y_1 > y_2) \wedge \text{at } l_0$

We can show (as is the case above) that $\varphi''$ and $\psi''$ satisfy the three conditions of the EVNT rule, so that $\varphi'' \supset \Diamond\psi''$, i.e.:

$$3.\ [\varphi(y_1,y_2,n+1) \wedge (y_1 > y_2) \wedge \text{at } l_1] \supset \Diamond[\varphi(y_1,y_2,n+1) \wedge (y_1 > y_2) \wedge \text{at } l_0]$$

From here the proof of case B1 proceeds:

4. $[\varphi(y_1,y_2,n+1) \wedge (y_1 > y_2) \wedge \text{at } l_1] \supset \Diamond\varphi(y_1,y_2,n)$
... by 2, 3 and $\Diamond$ Concatenation rule[83]

5. $[\varphi(y_1,y_2,n+1) \wedge (y_1 > y_2)] \supset \Diamond\varphi(y_1,y_2,n)$
... by 1, 2, 4 and propositional reasoning

This concludes a proof of case B1.

The proof of case B2 is symetrical to case B1. Because we have proved both cases B1 and B2, we may conclude that lemma B is true.

Lemma A and lemma B provide a foundation to prove the original theorem. Most of the steps in the proof require nothing more complex than propositional calculus. The proof of termination requires the natural number induction rule IND presented above.

Theorem proof:

6. $[\varphi(y_1,y_2,n+1) \wedge (y_1 \neq y_2)] \supset \Diamond\varphi(y_1,y_2,n)$
... Lemma B

7. $\varphi(y_1,y_2,n+1) \supset [(y_1 = y_2) \vee \Diamond\varphi(y_1,y_2,n)]$
... by propositional reasoning

8. $\varphi(y_1,y_2,n+1) \supset [\Diamond(y_1 = y_2) \vee \Diamond\varphi(y_1,y_2,n)]$
... by temporal and propositional reasoning

9. $\neg\varphi(y_1,y_2,0)$
... by propositional reasoning and the fact that the conjunction $(y_1>0) \wedge (y_2>0) \wedge (y_1+ysub2 \leq 0)$ is not possible.

10. $\varphi(y_1,y_2,0) \supset \Diamond(y_1 = y_2)$
... by propositional reasoning

11. $\varphi(y_1,y_2,n) \supset \Diamond(y_1 = y_2)$
... by 8, 10 and IND

12. $\exists n.\varphi(y_1,y_2,n) \supset \Diamond(y_1 = y_2)$
... by $\exists$ insertion rule[83]

13. $[\text{at}(l_0,m_0) \wedge (y_1,y_2) = (x_1,x_2) > 0] \supset \exists n.\varphi(y_1,y_2,n)$
... by taking $n = x_1 + x_2 > 0$

Now, assuming that $y_1 = y_2$ for all the possible transitions it is possible to show that:

$$14.\ (y_1 = y_2) \supset \Diamond[\text{at}(l_2,m_2) \wedge (y_1 = y_2)]$$

Using 12, 13, 14 and the $\Diamond$ Concatenation rule[83] results in:

15. $[\text{at}(l_0,m_0) \wedge (y_1,y_2) = (x_1,x_2) > 0] \supset \diamond[\text{at}(l_2,m_2) \wedge (y_1{=}y_2)]$

This proposition together with lemma A and the temporal rule that says that $\diamond(w_1 \wedge w_2) \supset (\diamond\, w_1 \wedge \diamond w_2)$ gives:

$$[\text{at}\ (l_0,m_0) \wedge (y_1,y_2) = (x_1,x_2)] \supset \diamond[\text{at}\ (l_2,m_2) \wedge y_1 = \gcd(x_1,x_2)]$$

Which is the theorem we wanted to prove.

It is interesting to notice that, although programs in temporal logic can be represented graphically, it is quite possible to prove theorems about programs without actually having to draw a graph. Manna and Pnueli rely on a labeled textual representation of the program in their proof of GCD. Actually drawing a graph of this program would not have made the proof any different.

### 5.3. A Formal Specification

Mediators are formally specified by describing the sequence control of language constructs and by giving the enabling conditions and state transformations of operations in the language. Temporal logic axiom schemas are used to specify some constructs in the language that share a common control abstraction. These serve as templates for all transitions having the corresponding flow of control. The actual axioms for specific operations are obtained by filling in the placeholders in these schemas, namely the enabling condition and the state transformation. Other language constructs that have a unique control abstraction are specified directly by temporal logic axioms.

We use Manna and Pnueli's temporal language [84]. Their language is a first–order language over a fixed domain and includes four temporal operators : However, we do not use their convention of explicitly specifying the values of all variables in the axioms. Instead, we

include only variables whose values change during the transition.

The current state of a mediated object is modeled as a tuple: $\Sigma = <\overline{ymed}, \overline{yproc}, \overline{yobj}, \bar{\pi}, \bar{\rho}, sp\_ptr, \bar{s}>$. Program variables are partitioned into three sets: local variables of the mediator ($\overline{ymed}$), local variables and parameters of object procedures ($\overline{yproc}$) and encapsulated data of the object ($\overline{yobj}$). $\overline{ymed}$ and $\overline{yproc}$ are further partitioned into sets of variables local to individual procedures. Thus $ymed_i^P$ ($yproc_j^Q$) refers to the variable i (j) in the mediator (object) procedure P (Q). The notation is overloaded by using the subscripts *val*, *v–r* and *local* to refer to value parameters, value–result parameters and local variables of a procedure respectively.

The *location variables* $\pi^i$ point to locations $l_j^i$ within the mediator and keep track of the mediator's multiple threads of control. The *service–location variables* $\rho^i$ mark the locations of active requests in the object. These variables point to labels within object procedures. The location variable *sp_ptr* is used solely for the initialization and termination of the mediator and points to locations in the **init** and **term** blocks. The special value $\lambda$ is used to denote a null value for location and service–location variables. The vector $\bar{s}$ records the status of service requests in the mediator. Each pending request has a corresponding element $s^k$ , where k is the request key, that takes the values:

req — the service has been requested, but not yet recognized by the mediator;

pre — the mediator has recognized the request, but it has not yet started execution;

active — the mediator has started execution of the request, but it has not terminated;

term — execution has terminated, but the mediator has not recognized termination;

post — the mediator has recognized termination, the request has not yet been released;

mu — there is no request pending for the key value k.

We use the notation STATUS(id) to refer to $s^{id}$, the status of the request with **key** "id"

As the actual number of requests and active processes is variable, the vectors $\bar{\rho}$ and $\bar{s}$ are considered to be infinite vectors indexed by the unique service request key.

The start state of the mediator has all the program variables undefined, all the location and service–location variables with the null value $\lambda$, all status variables with the null value *mu*, and *sp_ptr* pointing to the start location of the initialization block.

In order to make our axioms clearer, we define a new temporal operator $\circ^M$ (Next state of the mediator). Manna and Pnueli use the sometime operator "$\diamond$" to reason about the next state of a particular process. However, this is not strong enough for our purposes. The mediator maintains multiple threads of control which are interleaved only during the guard evaluation process. Once an open guard is selected and fired, the statements in its guarded list are executed consecutively until this thread of control either terminates or reaches the next guard evaluation point. To enforce the consecutive (non–interleaved) execution of a block of statements in the mediator, we need to be able to talk about the next state of the mediator. For this reason the new temporal operator $\circ^M w$ is defined to mean that $w$ is true in the state resulting from the very next transition taken by the mediator as opposed to transitions taken by concurrently executing service routines. This can be written in terms of the state of the mediator and the basic temporal logic operators as:

$$\circ^M w \equiv [(\overline{ymed}=\bar{u}) \wedge (\bar{\pi}=\bar{\xi})] \; \mathcal{U} \; w$$

where $\bar{u}$ and $\bar{\xi}$ are global constants used to talk about the values of program and location variables in a previous state.

We use the notation $\bar{x}=\bar{\omega}_{\circ}[w^{i}=m]$ to denote that vector $\bar{x}$ has the value of vector $\bar{\omega}$ with the value of the $i^{th}$ element replaced by $m$. We use the notation *at* $l$ to mean that some element $\pi^{i}$ of $\bar{\pi}$ is equal to label $l$.

### 5.3.1. Modeling the Mediator's Control Abstraction

**Body** and **cycle** are the two language constructs that explicitly alter the flow of control in a program. We describe the effect of these constructs on program labeling through the use of textual schemas (see figure 5.4). We then formally specify the sequence control defined by these constructs by giving temporal logic axiom schemas.

When a mediated object is instantiated, the mediator begins executing its initialization block (**init ... end init**). This is a simple sequential program on the mediators local data

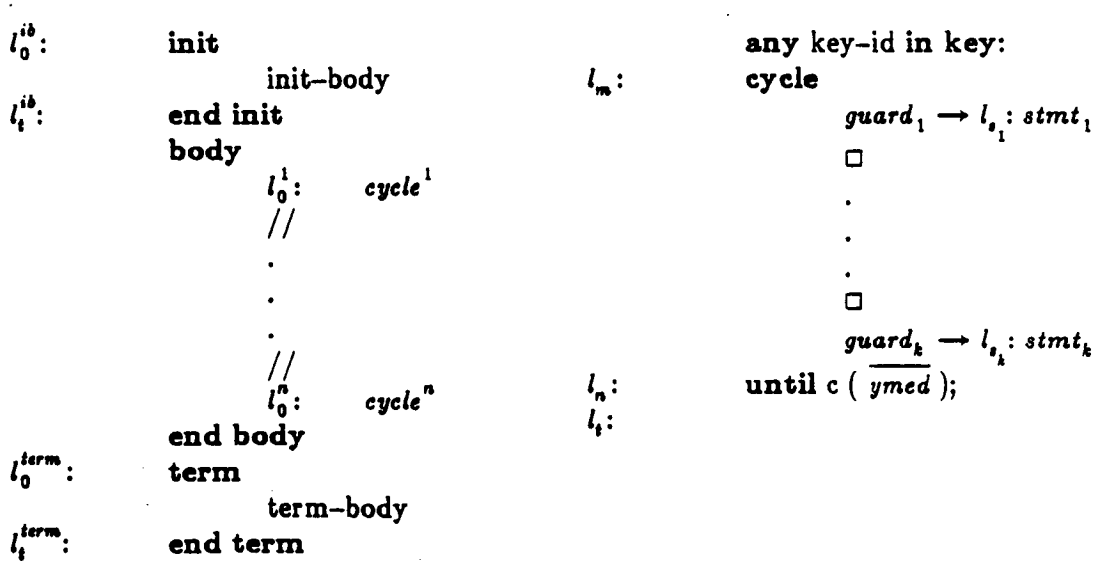



Figure 5.4. Textual Schemas for the labeling of the **body** and **cycle** constructs.

that can be modeled quite conventionally. The termination of the initialization block creates the initial state for execution of the mediator **body**. The significant change in state when the initialization block terminates is that $\bar{\pi}$ takes on an initial configuration as defined by the parallel guarded commands in the **body**. The initialization axiom expresses this transformation. In the initialization axiom, and in the termination axiom that follows, the labels refer to labels in figure 5.4.

$$\text{(1) Initialization:} \quad [(sp\_ptr{=}l_t^{ib}) \supset \circ^M[(sp\_ptr{=}\lambda) \wedge (\forall i)(1{\leq}i{\leq}n \supset \pi^i{=}l_0^i)]$$

The mediator **body** may also terminate if all the parallel guarded commands within it terminate. In this case, a termination block (**term ... end term**) will execute. The termination axiom describes the transformation from the mediator **body**'s multiple thread of control represented by $\bar{\pi}$ to the simple sequential control flow of the termination block.

$$\text{(2) Termination:} \quad [(\forall i)(1{\leq}i{\leq}n \supset (\pi^i{=}l_t^i))] \supset \circ^M[(sp\_ptr{=}l_0^{term}) \wedge (\forall i)(1{\leq}i{\leq}n \supset (\pi^i{=}\lambda))]$$

The axioms for the execution of the termination block are familiar and conventional.

The flow of control defined by **cycle** is more complicated and is specified in the form of a parameterized axiom schema. An axiom schema serves as a template for a group of language constructs that share a similar flow of control. The rule for a specific construct is built from a schema by substituting the precondition parameter (*pre*) and the postcondition parameter (*post*) with the appropriate values.

Schema 1 gives a template for the guard evaluation semantics. We define schema 1 for a transition from the **cycle** statement label $l_m$ to a statement list label $l_{s_i}$ which is associated with a particular guard of that **cycle** statement. This schema is used to describe how a sin-

gle guard is non-deterministically selected from the set of all open guards in the mediator and is fired. A guard is *open* if and only if some location variable $\pi^i$ points to a **cycle** statement immediately containing the guard and if the guard's condition is true in the current state. Schema 1 specifies weak fairness in guard evaluation, that is if the guard transition is infinitely often enabled, the transition will eventually be taken. The complete specification of the **cycle** construct also must include a specification of the termination condition, which is presented below.

**Schema 1** : Transitions corresponding to guard evaluation for a **cycle**.

$$\Box\Diamond[\text{at } l_m \wedge pre] \supset$$

$$\Diamond[\text{at } l_m \wedge (\bar{\pi}=\bar{\xi}) \wedge pre \wedge \circ^M[(\bar{\pi}=\bar{\xi}\circ[\xi^i=l_{t_i}]) \wedge post]]$$

{If the transition is enabled infinitely often in the future, then it will eventually be taken.}

The guard evaluation process also includes receiving new requests. A client may send a request to the mediator at any time during the mediator's activity. Reception of a request from a client with the key $i$ causes STATUS($i$) = *mu* to become STATUS($i$) = *pre*, and client $i$'s job descriptor to become available to the mediator. Because it is undesirable to have either the STATUS vector or the set of job descriptors in $\overline{ymed}$ change independently during mediator computations, the mediator may receive requests only under the same circumstances that it may evaluate guards.

This is specified by an axiom of the following form applied to every **cycle** construct in the program:

(3) [at $l_m$ $\wedge$ ($\exists$: i **in key:** *a request from* i *has arrived* $\wedge$ STATUS(i) = mu $\wedge$ i = u)] $\supset$
$\Diamond$ [at $l_m$ $\wedge$ STATUS(u) = pre]

This axiom allows for a delay in receiving requests and limits their reception to times when

guards may be evaluated.

The termination of service operations creates a similar situation. The service operation, once initiated executes independently of the mediator (see section 5.3.3). When an operation executed on behalf of client $i$ reaches its last statement and terminates, STATUS($i$) must change from *active* to *term*. Again, to keep the mediator state constant during mediator computatations, we must constrain when this change may occur. There are essentially two cases to consider. Service operations initiated by an **exec** command terminate while the mediator is blocked. These may be handled by the axioms of the **exec** statement presented below. Service operations that were initiated by a **spawn** terminate independently. The mediator delays recognizing the termination until a time when guards may be evaluated:

$$\text{(3a) } [\text{at } l_m \wedge (\exists \text{i: i in key: } \rho^i = l_t^P \ \&\ \text{STATUS(i)} = \text{active} \wedge \text{i} = \text{u})] \supset \\ \Diamond\, [\text{at l sub m} \wedge \text{STATUS(u)} = \text{term}]$$

In this axiom $l_m$ labels a **cycle** command, $l_t^P$ is the last statement in service routine $P$ and $\rho^i$ is a location counter for the service routine executed on client $i$'s behalf. More of the semantics of the **spawn** command is given below.

Schema 2 is a stronger axiom that enforces the consecutive execution of statements in the associated statement list of a guarded command.

**Schema 2** : Transitions corresponding to operations in st–list of **cycle**.

$$[\text{at } l_m \wedge (\bar{\pi}=\bar{\xi}) \wedge pre] \supset o^M[(\bar{\pi}=\bar{\xi}\circ[\xi^i=l_n]) \wedge post]$$

{If the transition is enabled in the current state, then it will be the very next transition taken by the mediator.}

The construct defined in this paper typically would be embedded in a conventional programming language. Additional axiom schemas would be needed to define control statements

in the host language, as per [81,94].

### 5.3.2. Semantics of operational statements

The previous section described the control abstraction of mediators in terms of temporal logic axiom schemas. These serve as templates for specifying the semantics of actual operations and permit the semantics to be expressed in a clear, concise and easy–to–read fashion. For those constructs that fit these schemas, we present their semantics by naming the schema involved and the values of the placeholders. Other constructs are specified directly with temporal logic axioms. English interpretations are used to highlight important aspects of the constructs.

The basic operation statements used in mediators include a **skip** statement and assignment statements, both of which may be specified in a conventional manner. The control flow of these statements is encapsulated in Schema 2.

The **skip** command may be specified as:

(4) **Construct:** $l_m$: **skip**; $l_n$:
**Schema used :** Schema 2
**Parameters:** pre: true
post: true

Assignment statements are specified as:

(5) **Construct:** $l_m$: $\overline{ymed} := h(\overline{ymed})$; $l_n$:
**Schema used :** Schema 2
**Parameters:** pre: $\overline{ymed}=\bar{u}$
post: $ymed=h(\bar{u})$

Local mediator procedure calls may be specified as in [64]. In the axiom that follows, $\bar{a}_{val}$ is a vector of actual values for call–by–value parameters. $\bar{a}_{v\rightarrow r}$ is a vector of actual values

for value–result parameters. The parameters passed into local routines will be from $\overline{ymed}$. We have ignored the possibility of side effects to simplify the axiom somewhat. Side–effects are dealt with in[64]. The label $l_0^P$ is the first statement in subroutine P. The element L in the axiom refers to a stack of location variables, with $|L|$ denoting the size of this stack and top(L) its top element[64]. This allows us to deal with recursive calls and to "remember" return locations. The variables $\overline{ymed}^P$ refers to routine P's parameters and local variables.

(6a) **Language Construct:** $l_m$: P $(\bar{a}_{val}, \bar{a}_{v-r})$; $l_n$:

$$[\text{at } l_m \wedge (|L|=s) \wedge (\overline{a}_{val}=\bar{u}_1) \wedge (\overline{a}_{v-r}=\bar{u}_2) \wedge (\overline{ymed}=\bar{u}) \wedge (\bar{\pi}=\bar{\xi})] \supset$$

$$o^M[(\bar{\pi}=\bar{\xi} \circ [\xi^i=l_0^P]) \wedge (|L|=s+1) \wedge (\text{top}(L)=l_n)$$

$$\wedge (\overline{ymed}=\bar{u} \circ [\overline{ymed}^P_{val}=\bar{u}_1; \overline{ymed}^P_{v-r}=\bar{u}_2; \overline{ymed}^P_{local}=\text{undefined}])]$$

The last component of this axiom that refers to elements of $\overline{ymed}^P$ expresses parameter passing.

In the following axiom that defines the return of routine P, the label $l_t^P$ is the last statement of the routine after which the routine will terminate.

(6b) $[\text{at } l_t^P \wedge (\text{top}(L)=l_n) \wedge (|L|=s) \wedge (\overline{ymed}=\bar{u}) \wedge (\bar{\pi}=\bar{\xi}) \wedge (\overline{ymed}^P_{v-r}=\bar{u}_1)] \supset$

$$o^M[(\bar{\pi}=\bar{\xi} \circ [\xi^i=l_n]) \wedge (|L|=s-1) \wedge (\overline{ymed}=\bar{u} \circ [\bar{a}_{v-r}=\bar{u}_1])]$$

The terms in this axiom that refer to $\overline{ymed}^P_{v-r}$ and $\bar{a}_{v-r}$ capture the return of value–result parameters.

### 5.3.2.1. Mediator Control Statements

Mediator guarded commands are considerably different from the original definition of guarded commands[32]. A mediator guarded command does not abort if none of its guards

evaluates as true, rather it delays until some guard becomes true (*cf.*[18]). Some mediator guards effect a change in request status when they fire. The guard evaluation semantics also determines the interleaving of statement execution. These characteristics allow guarded commands to be used to build complex synchronization schemes that depend on the requests present and the state of the mediator. The labels in the guard evaluation axioms that follow correspond to those of the **cycle** statement in figure 5.4.

The semantics of a guard which is a simple Boolean condition $c(\overline{ymed})$ is given by Schema 1 with the parameters having the values:

(7) **Construct:** $l_m : c(\overline{ymed}) \rightarrow l_{n_i}$:
**Schema used :** Schema 1
**Parameters:** pre: $c(\overline{ymed})$
post: true

Similarly, the semantics of a guard which is a status test paired with a Boolean condition is:

(8) **Construct:** $l_m$ : **req**(id);$c(\overline{ymed}$,id) $\rightarrow l_{n_i}$:
**Schema used :** Schema 1
**Parameters:** pre: (STATUS(id) = req) **cand** $c(\overline{ymed}$,id)
post: (STATUS(id) = pre)

For a **term** guard this is:

(8a) **Construct:** $l_m$ : **term**(id);$c(\overline{ymed}$,id) $\rightarrow l_{n_i}$:
**Schema used :** Schema 1
**Parameters:** pre: (STATUS(id) = term) **cand** $c(\overline{ymed}$,id)
post: (STATUS(id) = post)

Guarded commands prefixed by a key selector (**any** key_id **in key:**) may be handled by adding a quantifier to the enabling condition, and binding the key_id to an enabling value.

(9) **Construct:** $l_m$: **any** key–id **in key:** **req**(key–id);c($\overline{ymed}$,key–id) $\rightarrow$ $l_{s_i}$:

$$(\forall k\text{: k } \textbf{in key:}\ \underline{\Box\Diamond}\ [(\text{at } l_m \wedge (\text{STATUS(k)} = \text{req})\ \textbf{cand}\ \underline{c(\overline{ymed}, k)} \supset \Diamond(\text{ at } l_m \wedge \underline{(\overline{ymed} = \overline{u})} \wedge (\text{STATUS(k)} = \text{req})\ \textbf{cand}\ c(\overline{ymed}, k) \wedge o^M((\overline{ymed} = \overline{u}) \circ [\text{key\_id} = k]) \wedge (\text{STATUS(k)} = \text{pre})))]$$

This axiom describes the same flow of control as in schema 1, e.g. a guard that is infinitely often enabled will eventually be fired. We have not used schema 1 here because of the key binding that occurs in this axiom. A single guard of the form

**req**(key–id);c($\overline{ymed}$,key–id)

in a **cycle** statement that is prefixed by **any** key_id **in key:** describes a set of transitions, one for each value in the range described by the **key** field of the job descriptor. The axiom is written so that if the guard bound to a particular value of *k* in the range of **key** is fired, the mediator variable *key_id* is assigned that *k* on the transition to the associated statement list $l_{s_i}$:

If the guard contains a **term** status test, this rule is applied:

(9a) **Construct:** $l_m$: **any** key–id **in key:** **term**(key–id);c($\overline{ymed}$,key–id) $\rightarrow$ $l_{s_i}$:

$$(\forall k\text{: k } \textbf{in key:}\ \underline{\Box\Diamond}\ [(\text{at } l_m \wedge (\text{STATUS(k)} = \text{term})\ \textbf{cand}\ \underline{c(\overline{ymed}, k)} \supset \Diamond(\text{ at } l_m \wedge \underline{(\overline{ymed} = \overline{u})} \wedge (\text{STATUS(k)} = \text{term})\ \textbf{cand}\ c(\overline{ymed}, k) \wedge o^M((\overline{ymed} = \overline{u}) \circ [\text{key\_id} = k]) \wedge (\text{STATUS(k)} = \text{post})))]$$

Quantification is used in this axiom as in axiom (9). The only difference between these axioms is the status of the request that will satisfy the guard.

The **otherwise** guard also fits Schema 1. Its precondition is that the conjunction of all the other guards (as defined above) in the same **cycle** is *false*. The postcondition is *true*.

Guarded command termination may occur when the final **until** statement of the **cycle** statement is reached. The **cycle** terminates (passes to label $l_t$) when the exit condition is

true, and returns to the start of the **cycle** otherwise. Labels in this case correspond to those on the **cycle** statement in figure 5.4. This axiom describes the termination of a **cycle**:

$$(10)\ [\text{at } l_n \wedge (\bar{\pi}=\bar{\xi})] \supset \circ^M[(\neg c(\overline{ymed}) \wedge (\bar{\pi}=\bar{\xi}\circ[\xi^i=l_m]))\vee (c(\overline{ymed}) \wedge (\bar{\pi}=\bar{\xi}\circ[\xi^i=l_t]))]$$

It should be noted that evaluating the exit condition may not have side effects.

### 5.3.3. Mediator Service Statements

Three special commands within the mediator (**exec, spawn** and **release**) allow it to activate operations on the encapsulated resource and to return the results of operations to clients. None of the other mediator statements directly affect the encapsulated resource. **Exec** and **spawn** both create processes to perform service operations on the encapsulated data. The execution of an **exec** command causes the mediator to block until that operation has terminated.

(11) **Language construct:** $l_m$ : **exec**(id,P($\bar{a}_{val}$,$\bar{a}_{v-r}$)); $l_n$:
**Schema used :** Schema 2
**Parameters:** pre: (STATUS(id) = pre)
post: (STATUS(id) = term)

In addition, we need two axioms to describe the coupled execution of the service routine. In the first axiom (12), which describes a call on the service routine $P$, the location variable $\pi^i$ is given the temporary value $\beta$ to indicate that this thread of control of the mediator is blocked. Because the **exec** axiom above requires the $i^{th}$ thread to make the next mediator transition, this additional axiom enforces the blocking of the other threads of control. The symbol $\rho^{id}$ is a location variable for a process that executes a service operation for a request with the key *id*. It is set to the initial statement in the routine $P$: $l_0^P$.

$$(12)\ [\text{at } l_m \wedge (\text{STATUS(id)=pre})] \supset \Diamond[(\rho^{id}=l_0^P) \wedge (\text{STATUS(id)=active}) \wedge (\pi^i=\beta)]$$

The second additional axiom (13) describes what happens when the execution of the service routine terminates and returns.

$$(13)\ [(\rho^{id}=l_t^P) \wedge (\pi^i=\beta)] \supset \Diamond[(\pi^i=l_n) \wedge (\text{STATUS(id)=term})]$$

In this axiom the label $l_t^P$ is the exit label of the service routine. The axiom states that once the final statement in the service routine has been reached, eventually the status of the served request (STATUS (id)) will become *term*, and the mediator will resume execution.

Together these additional axioms specify that the mediator remains blocked until $\pi^i$ again points to a program location. They also capture the start and termination of the service routine ($\rho^{id}$). These are analagous to the rules for a simple procedure call and return. These axioms do not include the parameter passing aspect of these statements. Instead, we refer the reader to [64] and to the discussion above of parameter passing for local mediator routine calls.

The axioms to describe the **spawn** statement are simpler because the mediator does not wait for termination when a **spawn** is executed:

(14) **Language construct:** $l_m$ : **spawn**(id,P($\bar{a}_{val}$,$\bar{a}_{var}$)); $l_n$ :
**Schema used :** Schema 2
**Parameters:** pre: (STATUS(id) = pre)
post: $(\rho^{id}=l_0^P) \wedge$ (STATUS(id)=active)

The termination of a spawned service does not directly affect the flow of control of the mediator, and for this reason it is not neccessary to add an additional axiom like axiom 12 to describe blocking the mediator. The post condition in axiom (14) describes the initiation of a process $\rho^{id}$ to execute a service for request *id*. The label $l_0^P$ is that of the initial statement in

service procedure $P$.

An additional axiom describes the termination of the service and a change in the status of the job serviced:

$$(15)\ [\rho^{id} = l_t^P] \supset \Diamond[\text{STATUS(id)=term}]$$

The label $l_t^P$ is again the last statement in service routine $P$. The axiom says that once the final statement in the service routine is reached, eventually the routine will return and that the status of request *id* will become *term*.

Because the semantics of every statement in the mediator except the **cycle** statement requires the following statement to be executed next, the transition described in axiom (15) will only occur when guards may be evaluated.

In order to use the **spawn** and **exec** statement semantics in a mediator verification, we make the assumption that all service routines terminate. A complete verification of a mediated object would require demonstrating termination of the service routines using conventional proof techniques.

The **release** statement returns the results of service execution to the client. A client that has requested a service from the mediator blocks until the mediator "releases" it:

(16) **Language construct:** $l_m$ : **release**(id) ; $l_n$:
**Schema used :** Schema 2
**Parameters:** pre: (STATUS(id) = term) $\vee$ (STATUS(id) = post)
post: (STATUS(id) = mu) $\wedge$ release_job

We do not present the semantics for the returning of result parameters to the client but use the term *release_jobs* to refer to them. The semantics for this from the client's point of view is similar to that of procedure calls.

## 5.4. A Sample Verification of a Mediator

Verification of a mediated object consists of two parts. The mediator is shown to satisfy its synchronization and scheduling requirements. This requires the assumption that service operations, once initiated, do terminate. The operations are then verified to show that they provide the specified service. This approach is possible because the mediator and the encapsulated resource operate in disjoint data spaces.

As an example, we formally specify the synchronization requirements of a *reader_writer* mediator and briefly sketch its proof. The proof relies on program axioms extracted from the solution using the rules given in the previous section and on a temporal logic proof system (as in [84]).

The properties of the mediator that we might wish to verify are the invariance (safety) properties of partial correctness and mutual exclusion and the eventuality (liveness) properties of accessibility and liveness. The mutual exclusion requirement of the readers–writers problem is that access of the resource to readers and writers be mutually exclusive. This can be stated in temporal logic terms as:

(a) $\Box\,[(\textit{reader\_count}=0) \supset \neg\,(\exists \text{k: k in } \textbf{key}\text{: (STATUS(k) = active)} \wedge \text{(job(k).service = read))}]$

(b) $\Box\,[\neg\,(\exists \text{k: k in } \textbf{key}\text{: (STATUS(k) = active)} \wedge \text{(job(k).service = write))} \wedge (\textit{reader\_count}>0))]$.

(c) $\Box\,[0 \leq (\text{Nk: k in } \textbf{key}\text{: (STATUS(k) = active} \wedge \text{job(k). service = write))} \leq 1]$

The first assertion states that if *reader_count* = 0, no reader can be active. This assertion is proved by showing that *reader_count* is incremented whenever a new reader is activated and is decremented whenever an active reader terminates. The second assertion states that no writer can be active if *reader_count* is non–zero. This follows trivially from the semantics of

```
init
        reader_count := 0
end init
body
   any i in key:
   l: cycle
          req(i); job(i).service = write ->
               l1: cycle
                  reader_count = 0 ->
                       lexec: exec(i, write (job(i).writeprm));
                       lrelease: release(i);
               le1 : until true
          □
          req(i); job(i).service = read ->
               l2: reader_count := reader_count + 1;
               lspawn: spawn(i, read (job(i). readprm));
        le: until false
   //
   any i in key:
   m: cycle
          term(i); job(i).service = read ->
               m1: reader_count := reader_count - 1;
               mrelease: release(i);
        me: until false
lt:mt: end body
```

Figure 5.5. Labeled Reader_Writer Mediator.

the guard controlling the activation of a writer. Mutual exclusion of these operations is thus enforced by the judicious use of the counter *reader_count*. The third assertion states that there is never more that one write operation active at a time.

We will now prove the first assertion (a) to demonstrate how the temporal logic specification may be used to verify a mediator. The proof will follow the framework developed by Manna and Pnueli in[82]. Their method is a generalization of the *intermittent assertion* method[85]. In order to prove a program invariant, such as (a) above, it is neces-

sary to show that

- the assertion $\psi$ is true in the initial state of the program:
  e.g.: $\varphi(\overline{ymed}) \supset \psi$
- the assertion $\psi$ remains true for all possible mediator transitions.
  e.g.: for every transition in the mediator, if the assertion $\psi$ is true before the transition, it remains true afterwards.

If these can be demonstrated, then it can be inferred that

$$\varphi(\overline{ymed}) \supset \Box\, \psi.$$

Before we begin proving assertion (a), it is useful to demonstrate that certain parts of the mediator define critical sections that can be treated as a whole in our proof. These are:

$$L = \{l_{exec}, l_{release}\} \qquad L_2 = \{l_2, l_{spawn}\} \qquad M = \{m_1, m_{release}\}.$$

Using this notation *at L* means

$$\text{at } l_{exec} \vee \text{at } l_{release}.$$

Since L and $L_2$ both are locations of $\pi_1$, it is trivially true that

$$\Box \neg [\text{ at } L \wedge \text{at } L_2].$$

Proving these are critical sections requires us to prove the invariant:

$$\text{(d)}\ \ \Box \neg [(\text{at } L \vee \text{at } L_2) \wedge \text{at } M]$$

Since this is an invariant, the method we outlined above may be used. That is we must show:

(d1) assertion d is true initially
(d2) assertion d remains true for all possible transitions

The initial state of the mediator is:

$\bar{\pi} = \{l, m\}$, *reader_count* = 0, ($\forall$k: k in key: STATUS(k) = mu)
... prove by simple sequential proof of the **init** block and axiom (1).

Since in this state control is $\neg$ [at L $\vee$ at $L_2$ $\vee$ at M], assertion (d) is clearly true in this state. We have shown (d1).

Now we need to show that (d) holds for all possible transitions. In fact, Manna and Pnueli point out that only those transitions that can affect the value of an invariant assertion really must be considered[82]. In this case those transitions are:

(i): $l \rightarrow l_2$;
(ii): $l_1 \rightarrow l_{exec}$;
(iii): $m \rightarrow m_1$;
(iv): $l_{release} \rightarrow l_{e_1}$;
(v): $l_{spawn} \rightarrow l_e$;
(vi): $m_{release} \rightarrow m_e$.

To prove that assertion (d) holds, we assume it is true before each of these transitions, apply the transition and demonstrate that assertion (d) still holds.

Formally given a statement that fits schema 2, such as transitions (iv – vi), we must show that:

$$[(\text{at } l_n \wedge pre \wedge \neg [\text{ (at L} \vee \text{at } L_2) \wedge \text{at M}]) \supset (\text{at } l_n \wedge \neg [(\text{at L} \vee \text{at } L_2) \wedge \text{at M}])]$$

to prove that this holds for transition (iv) we must show that:

$$[(\text{at } l_{release} \wedge \neg [\text{ (at L} \vee \text{at } L_2) \wedge \text{at M}]) \supset (\text{at } l_{e_1} \wedge \neg [(\text{at L} \vee \text{at } L_2) \wedge \text{at M}])]$$

The proof that (d) holds for transition (iv):

| | |
|---|---|
| at $l_{release} \supset$ at L | ... definition of at L |
| at L $\supset \neg$ at M | ... assertion (c) in precondition |
| at $l_{release} \supset o^M$ at $l_{e_1}$ | ... by axiom (16) |
| [at $l_{release} \wedge \neg$ at M] $\supset o^M$[ at $l_{e_1} \wedge \neg$ at M] | ... propositional reasoning |
| at $l_{e_1} \wedge \neg$ at M | ... by making transition (iv) |
| $\neg$ [(at L $\vee$ at $L_2$) $\wedge$ at M])] | |

Transitions (v) and (vi) can be proved the same way.

Proving the invariant (d) is maintained in transitions (i–iii) is more difficult. To prove the invariant holds for transition (ii): $l_1 \rightarrow l_{exec}$; we must show that:

$$[(\text{at } l_1 \wedge \neg [(\text{at L} \vee \text{at } L_2) \wedge \text{at M}]) \supset (\text{at } l_{exec} \wedge \neg [(\text{at L} \vee \text{at } L_2) \wedge \text{at M}])]$$

The difficulty here is that the precondition: at $l_1 \wedge \neg [(\text{at L} \vee \text{at } L_2) \wedge \text{at M}]$ is true independent from the value of [at M]. We must consider the two cases:

case 1: at M
case 2: $\neg$ at M

Let us attempt to prove the invariant (d) on the transition (ii) given case 1:

| | |
|---|---|
| at M | ... given in case 1 |
| at M $\supset$ at $m_1 \vee$ at $m_{release}$ | ... definition of at M |

This introduces two subcases to prove: at $m_1$ and at $m_{release}$:

| | |
|---|---|
| at $m_1$ | ... first subcase |
| at $m_1 \supset o^M$ at $m_{release}$ | ... by axiom (5) |

However, if being *at* $m_1$ implies that the very next mediator state must come from the transition from $m_1$ to $m_{release}$, then it is impossible for us to make transition (ii), hence a contradiction. We must conclude: $\neg$ at $m_1$. The second subcase leads to the same contradiction:

| | |
|---|---|
| at $m_{release}$ | ... second subcase |
| at $m_{release} \supset o^M$ at $m_e$ | ... by axiom (16) |

and we deduce:

$$\neg \text{ at } m_{release}.$$

From these two subcases we derive:

$$\neg \text{ at } m_1 \wedge \neg \text{ at } m_{release} \supset \neg \text{ at M.}$$

Now we can easily prove the invariant on transition (ii):

| | |
|---|---|
| at $l_1 \wedge \neg$ [(at L $\vee$ at $L_1$) at M] | ... assumed |
| $\neg$ at M | ... by case 1 proof |
| at $l_{exec}$ | ... transition (ii) |
| at $l_{exec} \supset$ at $L_1$ | ... definition of at $L_1$ |
| at $l_{exec} \wedge \neg$ at M | ... axiom (7) on $\pi_1$ does not affect $\pi_2$ |
| at $l_{exec} \wedge \neg$ [(at L $\vee$ at $L_2$) $\wedge$ at M])] | |

This proves the invariant over transition (ii). The same technique can be used for proving transitions (i) and (iii) preserve the invariant (d).

From proving the invariance initially (d1) and over these transitions (d2) we can deduce

$$(d) \quad \Box \neg [(\text{at L} \vee \text{at } L_2) \wedge \text{at M}]$$

The mutual exclusion we have proved here can also be proven using this method for any segment of code that does not contain a guard evaluation. It is a natural consequence of the fact that initially all the elements of $\bar{\pi}$ point to guard evaluations and that all the transitions but guard evaluation are governed by the strong temporal operator $\circ^M$.

The proof of (a) is, once again, an invariance proof. To prove (a) we must show that the assertion:

[(*reader_count*=0) $\supset \neg$ ($\exists$k: k **in key**: (STATUS(k) = active) $\wedge$ (job(k).service = read))]

holds initially (a1), and holds through all possible transitions of the program (a2).

As we stated in proving (d1), the initial state of the mediator is:

$$\bar{\pi} = \{l, m\},\ \textit{reader\_count} = 0,\ (\forall k\text{: k in key: STATUS(k)} = \text{mu})$$

Since *reader_count* = 0 and no jobs are active, (a1) is trivially true. Here we make a slight diversion to prove a useful lemma:

$$(\text{lemma a.1}) \quad \Box [\textit{reader\_count} \geq 0]$$

This is another invariance property. It is obviously true in the initial state. Since the value

of *reader_count* is only changed in two places, we need prove the invariance over only the transitions (i) $l_2 \rightarrow l_{spawn}$ and (ii) $m_1 \rightarrow m_{release}$.

The proof for transition (i) is quite simple. Since statement $l_2$ adds to *reader_count*:

$$[\text{at } l_2 \wedge \textit{reader\_count} \geq 0] \supset [\text{at } l_{spawn} \wedge \textit{reader\_count} > 0]$$

by the definition of addition. We have already shown in assertion (d) that if this transition is enabled, it will be taken.

The proof of transition (ii) is more complicated. We need to show that:

$$[\text{at } m_1 \wedge \textit{reader\_count} \geq 0] \supset [\text{at } m_{release} \wedge \textit{reader\_count} \geq 0]$$

This can be broken down into two cases:

(case 1) *reader_count* $> 0$ ;
(case 2) *reader_count* $= 0$.

The assertion is obviously true for case 1, since subtracting one from a non-zero number can not make it negative. But what of case 2? Consider that the only transition that can lead to being *at* $m_1$ is the transition $m \rightarrow m_1$. The precondition for this transition is that for an $i$ that is bound at $m_1$:

$$[\text{at } m \wedge \text{STATUS(i)} = \text{term} \wedge \text{job(i).service} = \text{read}].$$

After the transition for $m \rightarrow m_1$ we have:

$$[\text{at } m_1 \wedge \text{STATUS(i)} = \text{post} \wedge \text{job(i).service} = \text{read}].$$

by axiom (9a). The problem here is that no job(i) can obtain the state STATUS(i) = term without previously having been the parameter of an **exec** statement (**exec**(i,...)) or a **spawn** statement (**spawn**(i,...)). We will not prove this formally here, but it is a clear result of the fact that only three of the axioms describing mediators allow the STATUS(i) to become term

(axioms 3a, 13 and 15), and these describe the results of a service operation (initiated by **spawn** or **exec**) completing.

Since the read operations in our mediator are only initiated by a **spawn** at label $l_{spawn}$. This means that in order for the transition $m \rightarrow m_1$ to be enabled, the transition $l_2 \rightarrow l_{spawn} \rightarrow l_e$ must have occurred. Since this transition only increments *reader_count*, we know that: $\neg$ [at $m_1 \wedge$ *reader_count* $= 0$|. From this argument we can deduce (lemma a.1).

As a second useful lemma, we show that *reader_count* is equal to the number of read jobs that are either STATUS = active or STATUS = term whenever a guard evaluation occurs. That is:

(a.lemma2) $\Box$[(at l $\vee$ at $l_1$) $\wedge$ at m] $\supset$ *reader_count* =
(Nk: k **in key**: (STATUS(k) = active $\vee$ STATUS(k) = term)
$\wedge$ job(k). service = read)

Please note that

[*reader_count* = (Nk: k **in key**:
(STATUS(k) = active $\vee$ STATUS(k) = term) $\wedge$ job(k). service = read)|

implies assertion (a).

Lemma a.2 is obviously true in the initial state. We will prove lemma a.2 over all transitions by showing that if we start in a state [(at l $\vee$ at $l_1$) $\wedge$ at m] that any deterministic path of transitions that return us to such a state maintains the invariant. In the semantics of the mediator, a chain of transitions each of which is defined with the $o^M$ operator forms a deterministic path. The mediator program consists of three such deterministic paths:

($pl_1$): $l_{exec} \rightarrow l_{release} \rightarrow l_{e_1} \rightarrow l$;
(pl): $l_2 \rightarrow l_{spawn} \rightarrow l_e \rightarrow l$;
(pm): $m_1$ -> $m_{release}$ -> $m_e$ -> m.

We need to show that if:

*reader_count* =
(Nk: k **in key**: (STATUS(k) = active ∨ STATUS(k) = term) ∧ job(k). service = read)

in a state in which

$$[(\text{at } l \vee \text{at } l_1) \wedge \text{at } m]$$

and we transition out of that state, it will still be true when we return to such a state. Since $pl_1$, pl and pm define all such transitions, we need to show that the transitions

(i) $l_1 \rightarrow pl_1$;
(ii) $l \rightarrow pl$;
(iii) $m \rightarrow pm$

maintain the assertion. Lemma a.2 is, in effect, a loop invariant. This method is supported by the Single Path Rule described in [82].

Lemma a.2 is trivially true over path $pl_1$ (transition i) as no element in that path alters the value of *reader_count* or changes the status of any read request.

We start transition (ii) with:

[(at l) ∧ (*reader_count* = (Nk: k **in key**: (STATUS(k) = active ∨ STATUS(k) = term)
∧ job(k). service = read))]

Since the transition described by pl increments *reader_count* at $l_2$, activates one read operation at $l_{spawn}$ and has no effect on any other job, the invariant holds at the end of pl (return to *at l*). The invariant holds over transition (ii).

The proof of transition (iii) is similar. We start with:

[(at m) ∧ (*reader_count* = (Nk: k **in key**: (STATUS(k) = active ∨ STATUS(k) = term)
∧ job(k). service = read))]

Since we assume we can fire the guard at m, it must be true that:

($\exists$k: k **in key**: (STATUS(k) = term) $\wedge$ job(k). service = read) ... by axiom (9a).

the effect of traversing path pm is to decrement *reader_count* by one and to remove one read job with STATUS = term. The path has no other effect, so when we return to m, lemma a.2 still holds.

We have proved that our assertion holds over all possible path transitions from the state [(at l $\vee$ at $l_1$) $\wedge$ at m]. We may deduce that lemma a.2 holds.

At this point we return to the proof of assertion (a). Rather than actually look at all possible mediator transitions in proving (a2), we may prune set of transitions to only those that affect elements in assertion (a). Those are transitions that assign to the *reader_count* variable and any transition that may change the status of a *read* request to of from STATUS(i) = active. These first two of these transitions are:

$$\text{(i) } l_2 \rightarrow l_{spawn}; \qquad \text{(ii) } l_{spawn} \rightarrow l_6.$$

We can prove the invariant (a) holds for transition (i) using lemma a.1:

[at $l_2$ $\wedge$ ((*reader_count*=0) $\supset$
$\neg$ ($\exists$k: k **in key**: (STATUS(k) = active) $\wedge$ (job(k).service = read)))] $\supset$
[at $l_{spawn}$ $\wedge$ ((*reader_count*=0) $\supset$
$\neg$ ($\exists$k: k **in key**: (STATUS(k) = active) $\wedge$ (job(k).service = read)))]

The proof for transition (i):

at $l_2$ $\wedge$ ((*reader_count*=0) $\supset$
$\neg$ ($\exists$k: k **in key**: (STATUS(k) = active) $\wedge$ (job(k).service = read)))
... assumed

*reader_count* $\geq$ 0
... lemma a.1

at $l_{spawn}$
... assume transition

at $l_{spawn}$ $\wedge$ *reader_count* > 0
... axiom (5), addition, lemma a.1

at $l_{spawn} \wedge \neg$ (*reader_count* = 0)

... propositional reasoning

at $l_{spawn} \wedge$ (*reader_count*=0) $\supset$
$\neg$ ($\exists$k: k **in key**: (STATUS(k) = active) $\wedge$ (job(k).service = read))

... (False $\supset$ x) = True

The proof of transition (i) feeds directly into the proof of transition (ii), that forms the critical section L that we proved mutually exclusive with assertion (d). Since above we have shown that

$$\text{at } l_{spawn} \supset \textit{reader\_count} > 0,$$

and since nothing in the **spawn** statement affects the value of *reader_count*, the implication

(*reader_count*=0) $\supset \neg$ ($\exists$k: k **in key**: (STATUS(k) = active) $\wedge$ (job(k).service = read))

remains trivially true.

We need to prove another transition in order to show the invariance of assertion (a). Since statement $m_1$ decrements *reader_count* we need to look at the transition (iii) $m_1 \rightarrow m_{release}$. Since statement $m_{release}$ is not defined on a read job with STATUS = active, there is no reason to consider the transition from that label. It is easier to see that the assertion holds for transition (iii) if we back up and consider that statement $m_1$ can only be reached by firing the guard at m. Since we assume we can fire the guard at m, it must be true that:

($\exists$k: k **in key**: (STATUS(k) = term) $\wedge$ job(k). service = read) ... by axiom (9a).

This means that:

(Nk: k **in key**: STATUS(k) = term $\wedge$ job(k). service = read)) $\geq 1$

And furthermore because of lemma a.2 we can see that if we are about to the guard at m is open for a transition to $m_1$ we know that in this case:

$$\textit{reader\_count} = 1 \supset \neg\,(\exists k\text{: k in key: (STATUS(k) = active)} \wedge \text{(job(k).service = read)}).$$

Firing the guard makes:

$$\text{at } m_1 \wedge \textit{reader\_count} = 1 \wedge \neg\,(\exists k\text{: k in key: (STATUS(k) = active)} \wedge \text{(job(k).service = read)}).$$

After transition (iii):

$$\text{at } m_{release} \wedge \textit{reader\_count} = 0 \wedge \neg\,(\exists k\text{: k in key: (STATUS(k) = active)} \wedge \text{(job(k).service = read)}).$$

which certainly satisfies invariant (a).

The case of transition (iii) :

$$\text{at } m_1 \wedge \textit{reader\_count} > 1$$

is less interesting. After the transition we get:

$$\text{at } m_{release} \wedge \textit{reader\_count} \geq 1$$

where assertion (a) holds trivially.

Now we have demonstrated that (a) holds in the initial state and that it continues to hold for all transitions in the mediator, hence we have proven that:

(a) $\Box\,[(\textit{reader\_count}=0) \supset$
$\neg\,(\exists k\text{: k in key: (STATUS(k) = active)} \wedge \text{(job(k).service = read)})]$.

The proof of assertion (b) that states mutual exclusion for write operations follows a similar outline. It is fairly easy to show that the deterministic path that can activate a write operation ($pl_1$ above) may only be entered with *reader_count* = 0. The proof of (c) can be derived directly from the fact that the exec statement is specified to block any further transitions in the mediator until the operation that it initiated terminates.

The second property that must be considered in a mediator proof is that of accessibility (a liveness property). In other words, if a service request for a read or a write has been received, then service for that particular request (i.e. the one bearing that specific key-identifier) should eventually be initiated. Formally,

(e) $(\forall k$: k **in key**: $((\text{STATUS}(k) = \text{req}) \wedge (\text{job}(k).\text{service} = \text{write})) \supset$
$\Diamond\, (\text{at } l_{exec} \wedge (\text{key\_id} = k)))$

(f) $(\forall k$: k **in key**: $((\text{STATUS}(k) = \text{req}) \wedge (\text{job}(k).\text{service} = \text{read})) \supset$
$\Diamond\, (\text{at } l_{spawn} \wedge (\text{key\_id} = k)))$

where $l_{exec}$ and $l_{spawn}$ are the labels of the **exec** and **spawn** statements in figure 5.5. The proof of these assertions requires the assumption that service operations always terminate and that they do not change the value of *reader_count*. It is then a standard exercise to apply the proof system to the program axioms to prove these assertions. Manna and Pnueli's EVNT rule[82] is essential to this proof.

## 5.5. Final Remarks

Once it was recognized that mediators did not exactly fit Manna and Pnueli's simplified model of a concurrent language[84], designing a model for mediators and writing a formal specification proved to be a straight-forward task. A number of different notations were considered, including Manna and Pnueli's graphical one. Schemas were developed mainly to allow us to present the semantics in a limited space[39].

Doing the formal specification provided a lot of information about the design of the mediated object, and in some cases led to changes in the design. For example, an earlier version of the mediator construct did not include exit conditions in the **cycle** statement. Instead, a **break** statement allowed an unconditional exit to the next statement after the

**cycle** that contained it. The **break** statement was rejected when we recognized that formally specifying such a construct in a direct manner was extremely difficult.

The elimination of the **break** statement in favor of an exit condition added to the **cycle** statement led to other changes. An earlier version of the design included two kinds of guard commands, a **cycle** statement that defined a looping guarded command, and a **when** statement that defined a "one time only" guarded command. Once an exit condition was added to the **cycle** statement, the **when** statement became redundant. It was merely equivalent to a **cycle** statement with an exit condition of *true*.

Temporal logic proved to be a very valuable tool in developing the mediated object. The fact that we could develop a formal specification allows us to have more confidence in the design, even without the practical experience of implementing and using the construct. The existence of an easy to use formal description should be useful to future implementors as an unambiguous statement of the design. It should also be useful to any future users who may wish to verify synchronization characteristics of the mediated objects they write.

# CHAPTER 6.

# SOME COMMENTS ON IMPLEMENTATION

Mediated objects have not yet been implemented, but we expect that implementing mediators should not present significant problems. Many of the components of the construct have been implemented in other languages. The main problem will be fitting the components together in an efficient manner.

There are several possible implementations for the mediator call mechanism. For example remote procedure calls could be applied [89]. A remote procedure call can be implemented as an exchange of messages between the client and mediator. The client sends a request message containing the name of the operation requested, its process identifier and parameters. It then waits to receive a reply, which will arrive when the mediator has released the operation. The mediator receives a request and creates a job descriptor. This is placed in the list of pending requests, becoming available for status tests. The job descriptor is destroyed when the mediator releases a job and returns results to the client. In the perception of the client process, a remote procedure call appears to be no different than a simple local procedure call. Mediator calls could also be implemented like Ada rendezvous[28] which are very similar to remote procedure calls in intent.

The **exec** and **spawn** statements require system support to initiate service for requests. This support may include creating a new system process and scheduling its execution on a free processor. Ramamritham and Keller's Sentinel Processes implement statements for coupled and uncoupled execution of this kind[98]. Their implementation could not be used directly as they operate with different parameters than our **exec** and **spawn** commands.

The evaluation of parallel guards appears to be a significant problem at first glance. The apparent distributed evaluation of guards from a number of parallel guarded commands does not immediately fit a familiar paradigm, and it appears that evaluating and reevaluating guards could be extremely inefficient.

The mediator body can be implemented in a direct, and somewhat simple–minded, way using a single process and a table of "active" guards. The active guard table would contain those guards corresponding to an element of the location vector $\bar{\pi}$ as described in the previous chapter, *e.g.* those guards at a control point. This essentially defines a jump table that can be used whenever the active flow of control reaches a guard evaluation. The table would need to be updated with each successful guard evaluation.

Guard evaluation does not need to be inordinately expensive. The special application of guards in mediators makes it possible to limit the number of guard reevaluations. After a guard evaluation, only certain events may change the value of the guards: the arrival of a new request, the termination of an active request or the execution of mediator statements after a guard has fired. If all guards have evaluated as false, there is no need to reevaluate the guards until either new requests arrive, or active requests terminate.

It is also possible to limit the number of guards considered during evaluation. The evaluation of guards containing status tests can be constrained in two ways. Status tests need only be evaluated for clients that are present in the mediators list of pending requests, since the value of any other status guard is automatically *false*. Application of fairness limits the evaluation of status tests for clients as well. These can be evaluated in the order of their arrival until an enabling guard is found.

The evaluation of pure Boolean guards cannot be limited this way. Fortunately, these are likely to be few in number. These also present a fairness problem. It is easy to apply a fair ordering criteria for requests based on time of arrival, but such criteria can not be applied to simple Boolean guards that may, without firing, become *true* and *false* repeatedly. Implementing weak fairness may require implementing event queues or counts so that these guards may be ordered.

The design of mediators is best suited to a system made up of distributed multiprocessor nodes, with one or several mediated objects installed at each node. Implementing mediators on such a system should be straightforward. Implementation of mediators on a uniprocessor is also possible using multiprogramming, but would probably be very inefficient. Mediators implemented on a distributed network of uniprocessors could work quite well. This could be accomplished by multiprogramming the mediated object on one node, or by allowing the mediator to exist on one node, and execute operations at remote nodes. The limiting factor would be the amount of object data that would need to be sent to the remote service nodes.

# CHAPTER 7.

# CONCLUSION

This paper has presented a preliminary proposal for a new language construct, the *mediated object*, that may serve as a useful tool in programming distributed embedded systems. Mediators allow direct programming of synchronization and scheduling and are able to directly use both information about a pending request and the present synchronization state. This makes mediated objects a powerful construct for synchronization and scheduling applications. At the same time, the design of mediated objects supports structured design of concurrent programs. We have also presented a temporal logic specification of the mediated object and indicated how this specification could be used to verify objects.

Mediated objects emphasize the principle of modularity in their design. In a mediated object the specification of the data abstraction representing a resource is designed and coded separately from the design of synchronization and scheduling for that resource. Because the resource is encapsulated within the mediated object, all uses of the resource are subjected to the synchronization and scheduling constraints implemented in the mediator. This separation and encapsulation both ensures a degree of protection for the resource and makes its design clearer.

The mediated object also allows for encapsulated concurrency. The mediator designer can specify as much or as little concurrency with the mediated object as desired, but still maintain encapsulation of the resource and its synchronization and ensure that the specified constraints will be enforced.

The flexibility of the mediated object is demonstrated by the wide variety of sample programs that we have been able to present. The mediated object is flexible both in terms of the kinds of synchronization and scheduling that it can implement, but also in terms of the way the mediator specifies those constraints.

Finally, we have presented a framework for verifying mediated objects. The flexibility of mediators in some senses makes trade–offs on the safety provided by more "protective" and less flexible concurrency structures such as those implemented in Argus[117]. This trade–off would not be acceptable if the object designer could not be confident that the object design met specifications. The temporal logic specification we have developed and the verification framework developed by[82] allow verification.

The newest elements of the mediator design include the design of parallel guards, the use of keys to mainpulate information about clients, the mapping of generic service names to actual operations within the mediator and the use of temporal logic as a primary language specification.

## 7.1. Directions for Further Research

The mediated object may provide a fertile source for future research in language implementation and design as well as research into verification and specification. It would be useful and instructive to complete an implementation of this construct in some suitable host language. An implementation in Pascal[59] could build on some of the work already done on implementing Path Pascal[23]. Another suitable host language may be C++[111] using classes as a basis for building mediated objects.

Mediators can implement non–preemptive scheduling schemes, but not preemptive ones. There are many systems applications in which the ability to do preemptive scheduling would be useful. In order to extend mediated onbjects to implement preemption it would be necessary to both include an interrupt mechanism and to provide a way to recover the state of the encapsulated resource. Recovery is needed because interrupted use of a software resource usually will leave that resource in an inconsistent state.

The implementation of a recovery mechanism is also essential in software fault–tolerance schemes. The mediated object could be extended to implement recovery blocks, much as Path Pascal was[105], or to implement conversations[26,58,100]. Besides a recovery mechanism, the extension to software fault–tolerance requires an exception handling mechanism that is totally lacking in the current design.

One feature of the mediator may be useful in developing a recovery block mechanism. The ability of the mediator to delay the return of results from a service after that service has terminated and to actually schedule when a **release** may occur may be directly applicable to programming coordinated termination for processes engaged in a conversation.

We have largely ignored the problems of hard deadline real–time programming in our discussion of the mediated object. This is a very important topic in programming for embedded systems, but also one that is extremely difficult to deal with. Considering the mediator in this context raises a number of interesting (and complex) questions that may offer a profitable area of research.

Finally, we have considered a number of issues concerning temporal logic as a specification and verification tool for mediated objects. This thesis was not intended to break new ground in this area, but rather to use existing temporal logic "technology" to support

our design. In the course of trying to apply temporal logic as presented for the rather simple programming languages discussed in[82,84], we discovered that these tools are not ready to be picked up and directly applied to languages that are richer in control structures and that operate in more complex domains. There is much that can be done to improve the usability of temporal logic.

## REFERENCES


1. Allchin, James E. and Martin S. McKendry. "Support for Objects and Actions in Clouds: Status Report", Tech. Report Georgia Institute of Technology, Atlanta, GA, Atlanta, GA, 1983.

2. ——. *Synchronization and Recovery of Actions.* **Preprint: 2nd ACM SIGACT–SIGOPS Symposium on Priciples of Distributed Computing** (Aug. 17–19, 1983).

3. Anderson, Thomas and John C. Knight. *A Framework for Software Fault Tolerance in Real–Time Systems.* **IEEE TOSE** (May 1983) vol. SE–9, no. 3, pp. 355–364.

4. Andler, Sten. *Predicate Path Expressions.* **6th Annual ACM Symposium on Principles of Programming Languages** (Jan. 29–31, 1979) pp. 226–236.

5. Andrews, Gregory R. *Parallel Programs: Proofs, Principles, and Practice.* **CACM** (March 1981) vol. 24, no. 3, pp. 140–146.

6. ——. *Synchronizing Resources.* **ACM TOPLAS** (Oct. 1981) vol. 3, no. 4, pp. 405–430.

7. ——. "*The Distributed Programming Language SR – Mechanisms, Design and Implementation*". **Soft. Pract. and Exper.** (1982) vol. 12, pp. 719–753.

8. Andrews, Gregory R. and Fred B. Schneider. *Concepts and Notations for Concurrent Programming.* **ACM Computing Surveys** (March 1983) vol. 15, no. 1, pp. 3–44.

9. Beichter, F., O. Herzog and H. Petzsch. "SLAN–4: Reference Manual and Design Rationale", Tech. Report: IBM Laboratory, Boeblingen, FRG, 1982.

10. Berg, H. K., B. E. Boebert and W. R. Franta. **Formal Methods of Program Verification and Specification.** Prentice Hall, Inc., Englewood Cliffs, NJ, 1982.

11. Best, Eike. "Relational Semantics of Concurrent Programs (With Some Applications)", Tech. Report: University of Newcastle Upon Tyne, Computing Laboratory, Newcastle Upon Tyne, UK, 1982.

12. Best, Eike and Brian Randell. *A Formal Model of Atomicity in Asynchronous Systems.* **Acta Informatica** (1981) vol. 16, pp. 93–129.

13. Birtwistle, G. M., O. J. Dahl, B. Myhrhaug and Kristen Nygaard. **Simula Begin.** Auerbach, Philadelphia, PA, 1973.

14. Bloom, Toby. *Evaluating Synchronization Mechanisms.* In: **Proc. 7th Symposium on OS Principles (Pacific Grove, CA. Dec 10–12).** ACM, New York, NY, 1979, pp. 24–32.

15. Brinch Hansen, Per. *Structured Multiprogramming.* **CACM** (July, 1972) vol. 15,

no. 7, pp. 574–578.

16. ——. *Concurrent Programming Concepts.* **ACM Computing Surveys** (Dec. 1973) vol. 5, no. 4, pp. 223–245.

17. ——. *The Programming Language Concurrent Pascal.* **IEEE TOSE** (1975) vol. SE–1, pp. 199–206.

18. ——. *Distributed Processes: A Concurrent Programming Concept.* **CACM** (Nov. 1978) vol. 21, no. 11, pp. 934–941.

19. ——. *Edison - A Multiprocessor Language.* **Soft. Pract. and Exper.** (1981) vol. 11, pp. 323–414.

20. Buckley, G. N. and Abraham Silberschatz. *An Effective Implementation for the Generalized Input–Output Construct of CSP.* **ACM TOPLAS** (April 1983) vol. 5, no. 2, pp. 223–235.

21. Campbell, Roy H. "Path Expressions: A Technique for Specifying Process Synchronization", Ph. D. Thesis: Univ. of Newcastle Upon Tyne, Newcastle Upon Tyne, UK, 1976.

22. Campbell, Roy H., Jeff Donnelly, Raymond B. Essick, Judith E. Grass, Dirk Grunwald, Pankaj Jalote and David A. McNabb. "The Embedded Operating System Project: Midyear Report, May 1984", Software Systems Research Group, University of Illinois at Urbana–Champaign, Dept. of Computer Science, Urbana, IL, 1984.

23. Campbell, Roy H. and Robert B. Kolstad. *An Overview of PATH PASCAL's Design.* **SIGPLAN Notices** (Sept. 1980) vol. 15, no. 9, pp. 13–14.

24. ——. *PATH PASCAL User Manual.* **SIGPLAN Notices** (Sept. 1980) vol. 15, no. 9, pp. 15–24.

25. ——. "A Practical Implementation of Path Expressions", Tech. report: Univ. of Illinois, Urbana–Champaign, Dept. Comp. Sci., Urbana, IL, 1980.

26. Campbell, Roy H. and Brian Randell. "Error Recovery in Asynchronous Systems", Tech. Report: Univ. of Illinois, Urbana–Champaign, Dept. Comp. Sci., Urbana, IL, 1984.

27. Courtois, P. J., F. Heymans and David L. Parnas. *Concurrent Control with Readers and Writers.* **CACM** (Oct. 1971) vol. 14, no. 10, pp. 667–668.

28. Defense, U. S. Department of. **Reference Manual for the ADA Programming Language: ANSI/MIL–STD–1815A–1983.** Springer–Verlag, New York, NY, 1983.

29. Deitel, Harvey M. **An Introduction to Operating Systems.** Addison–Wesley, Reading, MA, 1984.

30. De Millo, R. A., R. J. Lipton and A. J. Perlis. *Social Processes and Proofs of Theorems and Programs.* **CACM** (May 1979) vol. 22, no. 5, pp. 271–280.

31. Dijkstra, Edsger W. *Cooperating Sequential Processes.* In: **Programming Languages**, F. Genuys, ed. Academic Press, New York, NY, 1968.

32. ——. *Guarded Commands, Nondeterminacy and Formal Derivation of Programs.* **CACM** (Aug. 1975) vol. 18, no. 8, pp. 453–457.

33. ——. **A Discipline of Programming.** Prentice–Hall, Englewood Cliffs, NJ, 1976.

34. Dreisbach, T. A. and L. Weissman. *Requirements for Real–Time Languages.* In: **Lecture Notes in Comp. Sci.**. Springer Verlag, Berlin, FRG, 1977, pp. 298–312.

35. Feldman, Jerome A. *High_Level Programming for Distributed Computing.* **CACM** (June 1979) vol. 22, no. 6, pp. 353–367.

36. Filman, Robert E. and Daniel P. Friedman. **Coordinated Computing: Tools and Techniques for Distributed Software.** McGraw–Hill, New York, NY, 1984.

37. Good, Donald I., Richard M. Cohen and James Keeton–Williams. *Principles of Proving Concurrent Programs in GYPSY.* **6th Annual ACM Symposium on Principles of Programming Languages** ((San Antonio, TX. Jan. 29–31, 1979)) pp. 42–51.

38. Grass, Judith E. and Roy H. Campbell. *Mediators: A Synchronization Mechanism.* **to appear in : The 6th Int. Conf. on Distributed Computing Systems** (May 19–23, 1986).

39. Grass, Judith E., Vipin Swarup, Simon M. Kaplan and Roy H. Campbell. "A Temporal Logic Specification of the Mediator Construct: Extended Abstract", Extended abstract submitted to the 5th ACM SIGACT–SIGOPS Symposium on the Principles of Distributed Computing, University of Illinois at Urbana, Champaign; Urbana, IL, 1986.

40. Gray, Jim. *Notes on Data Base Operating Systems.* In: **Operating Systems, an Advanced Course. Lecture Notes in Computer Science, Vol. 60**, R.M. Graham and G. Seegmueller R. Bayer, ed. Springer–Verlag, NY, 1978, pp. 393–481.

41. Gries, David. **The Science of Programming.** Springer–Verlag, New York, NY, 1981.

42. Habermann, A. Nico and Dewayne E. Perry. **Ada for Experienced Programmers.** Addison–Wesley, Reading, MA, 1983.

43. Hehner, Eric C. R. *Predicative Programming.* **CACM** (Feb. 1984) vol. 27, no. 2, pp. 134–151.

44. Hewitt, C. E. and Russell Atkinson. *Specifications and Proof Techniques for Serializers.* **IEEE TOSE** (1979) vol. SE–5, no. 1, pp. 10–23.

45. Hoare, C. A. R. *An Axiomatic Basis for Computer Programming.* **CACM** (Oct. 1969) vol. 12, no. 10, pp. 576–583.

46. ——. *Procedures and Parameters: An Axiomatic Approach.* In: **Symposium on Semantics of Algorithmic Languages**, E. Engeler, ed. Springer–Verlag, Berlin, FRG, 1971, pp. 102–115.

47. ——. *Proof of Correctness of Data Representations.* **Acta Informatica** (1972) vol. 1, pp. 271–281.

48. ——. *Towards a Theory of Parallel Programming.* In: **Operating Systems Techniques**, C. A. R. Hoare and R. H. Perrott, ed. Academic Press, London, 1972, pp. 61–71.

49. ——. *Monitors: An Operating System Structuring Concept.* **CACM** (Oct. 1974) vol. 17, no. 10, pp. 549–557.

50. ——. *Communicating Sequential Processes.* **CACM** (Aug. 1978) vol. 21, no. 8, pp. 666–677.

51. ——. *Data Structures.* In: **Current Trends in Programming Methodology, Volume IV: Data Structuring**, Raymond T. Yeh, ed. Prentice Hall, Inc., Englewood Cliffs, NJ, 1978, pp. 1–11.

52. ——. *A Model for Communicating Sequential Processes.* In: **On The Construction of Programs**, R. M. McKeag and A. M. McNaughton, ed. Cambridge University Press, Cambridge, UK, 1980, pp. 229–243.

53. ——. *A Calculus of Total Correctness for Communicating Processes.* **Science of Computer Programming** (1981) vol. 1, pp. 49–72.

54. Holt, Richard C. **Concurrent Euclid, The UNIX® System and TUNIS.** Addison–Wesley, Reading, MA, 1983.

55. Horton, Kurt H. "A Fault–Tolerant Deadline Mechanism", MS Thesis, University of Illinois at Urbana–Champaign, UIUCDCS–R–79–998, Urbana, IL, 1979.

56. Howard, J. H. *Signalling in Monitors.* In: **Proc. 2nd Int. Conf. Softw. Eng..**, San Francisco, CA), 1976, pp. 47–52.

57. Intermetrics, Inc. "HAL/S Manual", Intermetrics, Inc., Cambridge, MA, 1975.

58. Jalote, Pankaj and Roy H. Campbell. *Recoverability of Actions and Atomicity.* **IEEE TOSE** (Jan. 1986) vol. SE–12, no. 1, pp. 59–68.

59. Jensen, Kathleen and Niklaus Wirth. **Pascal: User Manual and Report.** Springer–Verlag, New York, NY, 1974.

60. Jones, Cliff B. **Software Development: A Rigorous Approach.** Prentice/Hall International, Englewood Cliffs, NJ, 1980.

61. ——. *Specification and Design of (Parallel) Programs.* In: **Information Processing '83**, R.E.A. Mason, ed. Elsevier Science Publishers, B.V. (North Holland), 1983, pp. 321–332.

62. ——. *Tentative Steps Toward a Development Method for Interfering Programs.* **ACM TOPLAS** (Oct. 1983) vol. 5, no. 4, pp. 596–619.

63. Kaplan, Simon M. "Specification and Verification of Context Conditions for Programming Languages", Tech. Rep. UIUCDCS–R–85–1232 Dept. of Comp. Sci., University of Illinois at Urbana– Champaign, Urbana, IL, 1985.

64. ——. "Verification of Recursive Programs: A Temporal Proof Approach", Tech.

Rep. UIUCDCS-R-85-1207 Dept. of Comp. Sci., University of Illinois at Urbana-Champaign, Urbana, IL, 1985.

65. Kieburtz, Richard B. and Abraham Silberschatz. *Comments on "Communicating Sequential Processes"*. **ACM TOPLAS** (Oct., 1979) vol. 1, no. 2, pp. 218–225.

66. Kolstad, Robert B. "Distributed Path Pascal: A Language for Programming Coupled Systems", Ph. D. Thesis, Tech. Report: Dept. Comp. Sci., University of Illinois at Urbana-Champaign, UIUCDCS-R-83-1136, Urbana, IL, 1983.

67. Kolstad, Robert B. and Roy H. Campbell. "Path PASCAL User Manual", Tech. Report: Dept. of Comp. Sci. University of Illinois at Urbana-Champaign UIUCDCS-R-80-893, Urbana, IL, 1980.

68. Lamport, Leslie. *Proving the Correctness of Multiprocess Programs.* **IEEE TOSE** (March 1977) vol. SE-3, no. 2, pp. 125–143.

69. ——. *The 'Hoare Logic' of Concurrent Programs.* **Acta Informatica** (1980) vol. 14, pp. 21–37.

70. ——. *Specifying Concurrent Program Modules.* **ACM TOPLAS** (April 1983) vol. 5, no. 2, pp. 190–222.

71. Lamport, Leslie and Fred B. Schneider. *The 'Hoare Logic' of CSP, and All That.* **ACM TOPLAS** (April 1984) vol. 6, no. 2, pp. 281–296.

72. Lauer, H. C. and R. M. Needham. *On the Duality of Operating System Structures.* **ACM Op. Syst. Rev.** (April 1979) vol. 13, no. 2, pp. 3–19.

73. LeBlanc, Thomas J. *Programming Language Support for Real-Time Distributed Systems.* In: **Proc. International. Conference on Data Engineering**. The Computer Society Press, NY, New York, NY, 1984, pp. 371–376.

74. Leinbaugh, Dennis W. "High Level Specification and Implementation of Resource Sharing", Tech. Report: Ohio State University, OSU-CISRC-TR-81-3, Columbus, OH, 1981.

75. ——. "High Level Description and Synthesis of Resource Schedulers", Submitted to ACM '82: Ohio State University, Columbus, OH, 1982.

76. Liskov, Barbara H. *On Linguistic Support for Distributed Programs.* **IEEE TOSE** (May 1982) vol. SE-8, no. 3, pp. 203–210.

77. Liskov, Barbara H. and Robert Scheifler. *Guardians and Actions: Linguistic Support for Robust, Distributed Programs.* **ACM Tran. Prog. Lang. and Syst.** (July 1983) vol. 5, no. 3, pp. 381–404.

78. Luckham, David C. and Norihisa Suzuki. *Verification of Array, Record and Pointer Operations in Pascal.* **ACM TOPLAS** (Oct., 1979) vol. 1, no. 2, pp. 226–244.

79. McKendry, Martin S. "Language Mechanisms for Context Switching and Protection in Level Structured Operating Systems", Tech. Report: Dept. of Comp. Sci., University of Illinois at Urbana-Champaign, UIUCDCS-R-81-1078,

Urbana, IL, 1981.

80. Manna, Zohar. "Verification of Sequential Programs: Temporal Axiomatization", Tech. Rep. STAN-CS-81-877 Dept. of Comp. Sci., Stanford Univ., Stanford, CA, 1981.

81. Manna, Zohar and Amir Pnueli. "How to Cook a Temporal Proof System for Your Pet Language", Tech. Rep. STAN-CS-82-954 Dept. of Comp. Sci., Stanford Univ., Stanford, CA, 1982.

82. ——. *Verification of Concurrent Programs: Temporal Proof Principles.* In: **Logics of Programs: LNCS 131**, Dexter Kozen, ed. Springer- Verlag, New York, NY, 1982, pp. 200–252.

83. ——. "Verification of Concurrent Programs: A Temporal Proof System", Tech. Rep. STAN-CS-83-967, Dept. of Comp. Sci., Stanford University, Stanford, CA, 1983.

84. ——. *Verification of Concurrent Programs: The Temporal Framework.* In: **The Correctness Problem in Computer Science**, R. S. Boyer and J. S. Moore, ed. Academic Press, London, UK, 1983, pp. 215–273.

85. Manna, Zohar and R. Waldinger. *Is "Sometime" Sometimes Better Than "Always"? Intermittent Assertions in Proving Program Correctness.* **CACM** (Feb. 1978) vol. 21, no. 2, pp. 159–172.

86. May, David. *OCCAM.* **SIGPLAN Notices** (April 1983) vol. 18, no. 4, pp. 69–79.

87. Mitchell, J. G., W. Maybury and R. Sweet. "Mesa Language Manual, Version 5.0", Xerox Palo Alto Research Center, Rep. CSL-79-3, Palo Alto, CA, 1979.

88. Mizell, David. *Verification and Design Aspects of 'True Concurrency'.* **5th Annual ACM Symposium on Principles of Programming Languages** (Tucson, AZ. Jan 23–25, 1978) pp. 171–175.

89. Nelson, B. J. "Remote Procedure Call", Ph. D. Dissertation, Carnegie-mellon University, Pittsburgh, PA, 1981.

90. Oldehoeft, Arthur E. and Steven F. Jennings. *Dataflow Resource Managers and Their Synthesis from Open Path Expressions.* **IEEE TOSE** (May 1984) vol. SE-10, no. 3, pp. 244–257.

91. Ossefort, Marty. *Correctness Proofs of Communicating Processes: Three Illustrative Examples from the Literature.* **ACM TOPLAS** (Oct. 1983) vol. 5, no. 4, pp. 620–640.

92. Owicki, Susan and David Gries. *An Axiomatic Proof Technique for Parallel Programs.* **Acta Informatica** (1976) vol. 6, pp. 319–340.

93. ——. *Verifying Properties of Parallel Programs: An Axiomatic Approach.* **CACM** (May 1976) vol. 19, no. 5, pp. 279–285.

94. Owicki, Susan and Leslie Lamport. *Proving Liveness Properties of Concurrent Programs.* **ACM TOPLAS** (July 1982) vol. 4, no. 3, pp. 455–495.

95. Pagan, Frank G. **Formal Specification of Programming Languages: A**

**Panoramic Primer**. Prentice Hall, Inc., Englewood Cliffs, NJ, 1981.

96. Patil, S. S. "Limitations and Capabilities of Dijkstra's Semaphore Primitives for Co–ordination amongst Processes.", Project MAC, Computational Structures Group Memo 57, 1971.

97. Peterson, James L. and Abraham Silberschatz. **Operating System Concepts**. Addison–Wesley Publishing Co., Reading, MA, 1983.

98. Ramamritham, Krithivasan and Robert M. Keller. *Specifying and Proving Properties of Sentinel Processes*. In: **Proc. 5th Int. IEEE/ACM Soft. Eng. Conference.**, 1981, pp. 374–381.

99. ——. *Specification of Synchronizing Processes*. **IEEE TOSE** (Nov. 1983) vol. SE–9, no. 6, pp. 722–733.

100. Randell, Brian. *System Structure for Software Fault Tolerance*. **IEEE TOSE** (June 1975) vol. SE–1, no. 2, pp. 220–232.

101. Ritchie, Dennis M. and Ken Thompson. *The UNIX Timesharing System*. **CACM** (July, 1974) vol. 17, no. 7, pp. 365–375.

102. Rowe, L. A. *Programming Language Issues for the 1980's*. **SIGPLAN Notices** (Aug. 1984) vol. 19, no. 8, pp. 51–61.

103. Schlichting, Richard D. and Fred B. Schneider. *Understanding and Using Asynchronous Message Passing Primitives*. In: **Proc. Symp. Principles of Distributed Computing**. ACM, New York, NY, 1982, pp. 141–147.

104. ——. "Using Message Passing for Distributed Programming: Proof Rules and Disciplines", Tech. Rep. TR 82–491, Dept. of C.S., Cornell University, Ithaca, NY, 1982.

105. Schmidt, George Joseph. "The Recoverable Object as a Means of Software Fault Tolerance", MS Thesis: University of Illinois at Urbana–Champaign, Urbana, IL, 1983.

106. Shatz, Sol M. *Communications Mechanisms for Programming Distibuted Systems*. **Computer** (June 1984) vol. 17, no. 6, pp. 21–28.

107. Shaw, Alan C. **The Logical Design of Operating Systems**. Prentice–Hall, Inc., Englewood Cliffs, NJ, 1974.

108. Shields, M. W. *Adequate Path Expressions*. In: **Proc. Int. Symp. Semantics of Concurrent Computation, Lecture Notes in Computer Science, Vol. 70**. Springer–Verlag, Berlin, pp. 249–265.

109. Silberschatz, Abraham. *Communication and Synchronization in Distributed Systems*. **IEEE TOSE** (Nov. 1979) vol. SE–5, no. 6, pp. 542–546.

110. Stoy, J. E. **Denotational Semantics – The Scott–Strachey approach to Programming Language Theory**. MIT Press, Cambridge, MA, 1977.

111. Stroustrup, Bjarne. **The C++ Programming Language**. Addison–Wesley, Reading, MA, 1986.

112. Thompson, J. R. *The Use and Abuse of Formal Proofs.* **SIGPLAN Notices** (July 1983) vol. 18, no. 7, pp. 75–79.

113. Verjus, Jean-Pierre. *Synchronization in Distributed Systems: an Informal Introduction.* In: **Distributed Computing Systems**, Y. Paker and J.-P. Verjus, ed. Academic Press, London, 1983, pp. 3–22.

114. Wegner, Peter. *Programming With Ada: An Introduction by Means of Graduated Examples.* **SIGPLAN Notices** (Dec. 1979) vol. 14, no. 12, pp. 1–46.

115. Wegner, Peter and Scott A. Smolka. *Processes, Tasks, and Monitors: A Comparative Study of Concurrent Programming Primitives.* **IEEE TOSE** (July 1983) vol. SE–9, no. 4, pp. 446–462.

116. Wei, Anthony Y. "Real-Time Programming with Fault-Tolerance", Ph. D. thesis, Tech. Report: Dept. of Comp. Sci., University of Illinois at Urbana-Champaign UIUCDCS-R-81-1041, Urbana, IL, 1981.

117. Weihl, William and Barbara H. Liskov. *Specification and Implementation of Resilient, Atomic Data Types.* **SIGPLAN Notices** (June 1983) vol. 18, no. 6, pp. 53–64.

118. Williamson, Ronald and Ellis Horowitz. *Concurrent Communications and Synchronization Mechanisms.* **Soft. Pract. and Exper.** (Feb. 1984) vol. 14, no. 2, pp. 135–151.

119. Wirth, Niklaus. *Modula: A Language for Modular Multiprogramming.* **Software–Practice and Experience** (1977) vol. 7, pp. 3–84.

120. ——. *Toward a Discipline of Real-Time Programming.* **CACM** (Aug. 1977) vol. 20, no. 8, pp. 577–583.

121. Wolper, Pierre. *Specification and Synthesis of Communicating Processes Using an Extended Temporal Logic.* In: **Proc. 9th Ann. ACM Symp. POPL.** ACM, New York, NY, 1982, pp. 20–33.

122. Young, Stephen J. **An Introduction to Ada.** John Wiley and Sons, New York, NY, 1983.

# VITA

Judith Ellen Grass was born in Hartford, Connecticut in 1953. She received a B.S. degree magna cum laude in Modern Languages from Georgetown University, Washington, D.C. In 1977 she received an A.M. degree in Slavic Linguistics from the University of Illinois at Urbana–Champaign. In the course of earning these degrees she also attended summer classes in the U.S.S.R and Jugoslavia.

Through work in developing Russian language teaching materials on the PLATO system, she became interested in computer science. In 1982 she completed an M.S. in Computer Science at the University of Illinois. While working on her Ph. D. thesis, Ms. Grass spent two summers working as an intern student at I.B.M. in Poughkeepsie and Yorktown Heights, N.Y. She received her Ph.D. in Computer Science from the University of Illinois in 1986.

Ms. Grass is a member of the ACM and IEEE and is interested in concurrent processing, computer languages and software engineering issues.

| BIBLIOGRAPHIC DATA SHEET | 1. Report No. UIUCDCS-R-86-1266 | 2. | 3. Recipient's Accession No. |
|---|---|---|---|
| 4. Title and Subtitle<br>Mediators: A High-Level Language Construct For Distributed Systems | | | 5. Report Date<br>April 17, 1986<br>6. |
| 7. Author(s)<br>Judith Ellen Grass | | | 8. Performing Organization Rept. No. UIUCDCS-R-86-1266 |
| 9. Performing Organization Name and Address<br>Department of Computer Science, UIUC<br>1304 W. Springfield Avenue<br>Urbana, Illinois 61801 | | | 10. Project/Task/Work Unit No.<br>11. Contract/Grant No.<br>NASA NSG 1471 |
| 12. Sponsoring Organization Name and Address<br>NASA Langley Research Center<br>Hampton, Virginia 23665 | | | 13. Type of Report & Period Covered<br>Ph.D. Thesis<br>14. |

15. Supplementary Notes

16. Abstracts

This thesis describes the mediated object construct. Mediated objects support synchronization and scheduling for systems programming within distributed systems. Mediated objects are based on a resource view of systems, and fit within a programming methodology that emphasizes resource modularity, synchronization modularity and encapsulated concurrency.

A mediated object consists of an interface specification, a data abstraction construct (an object) and a separate mediator module that specifies synchronization and scheduling with the mediated object. The mediator displays many interesting features. These include: an adaptation of guarded commands; keys that allow requests to be examined and manipulated before they receive service; parallel guard execution; coupled and uncoupled modes of service execution.

The design of the mediated object construct is first presented informally with many programming samples. A temporal logic specification is also presented as a formal description of the construct. The temporal logic may be used for verifying mediated objects. A sample verification is included. Few practical languages have been specified with temporal logic. The specification provided helpful feedback during the development of the construct.

Finally, the thesis discusses a few aspects of implementation and offers suggestions for future research.

17. Key Words and Document Analysis. 17a Descriptors

Distributed Programming, Distributed Systems, Temporal Logic, Concurrency, Language Design

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

| 18. Availability Statement | 19. Security Class (This Report) | 21. No. of Pages |
|---|---|---|
| Unlimited | UNCLASSIFIED | 128 |
| | 20. Security Class (This Page) UNCLASSIFIED | 22. Price |

# A Physical Device Model for Path Pascal

Kevin B. Kenny

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, Illinois

# A Physical Device Model for Path Pascal

by

Kevin B. Kenny

University of Illinois at Urbana–Champaign

*DRAFT–Do not circulate*

**Abstract.** Path Pascal has proven itself a useful tool in simulating the behavior of multiprogramming systems, both in research and in teaching. One lack it has demonstrated is the ability to simulate the action of input–output devices in order to model operating systems on real machines. To rectify this lack, a system is presented that allows the Path Pascal programmer simulated access to a variety of I/O equipment, and a device model is presented which allows additional types of devices to be defined at will.

## 1. Introduction.

The Path Pascal programming language is designed to allow the user to experiment with the programming of multiprogramming systems. Its greatest use is in designing and simulating operating systems; for this purpose, however, the language itself is incomplete. One feature that the language lacks, by design, is any support for I/O devices.

This lack is rectified in the support software by providing a device simulator, written as a set of external objects that can be linked with Path Pascal programs. Through these objects, the user can define a set of peripherals, such as disk and tape drives, unit record equipment, and terminals, and allow the program to communicate with them.

These simulated devices are capable of operating in two major modes. In one of these, the program runs freely in a way that can be termed "pseudo–real time." In this mode, device requests are handled concurrently with the operation of the program and allowed to complete as rapidly as the underlying system's response time allows. The "pseudo–" designation reflects the fact that the Path Pascal program is itself running on a multiporgrammed system, and hence has other programs competing for the system's resources; true real–time simulation is impossible in such an environment.

The other mode may be termed "simulated–time" or "synchronous." In this mode, I/O requests are still processed concurrently as much as possible, but their terminations are scheduled with respect to the `wallclock` time in the Path Pascal system. Each device type being simulated embodies a model of that device's service time as requests

The work presented herein was funded in part by the National Aeronautics and Space Administration under Grant NSG–1471.

are presented to it, and the termination times therefore give some idea as to how the program would perform in real-time on an actual machine.

## 2. The Device Model.

Understanding the model for how a program communicates with a device is crucial to using the device simulator programs effectively. While somewhat simplistic, the model provides a reasonably accurate simulation of nearly all devices available today.

Each device is represented by a `device` object in the Path Pascal program. A `device` supports four fundamental operations: `open`, `close`, `do_io`, and `await_int`.

Each device on which I/O is to be performed must be initialized with an `open` request. This requests takes two arguments: a character string which is a shell command to start the device server, and a Boolean value which describes what is to happen with `SIGINT` signals from the operating system (We ignore the latter feature for now; see the description of the `tty` device for details).

The `open` request starts a server program that simulates the requested type of device, and initializes certain internal control information. Following this, `do_io` and `await_int` requests may be used to perform input from and output to the device.

Most I/O proceeds through the `do_io` request. Its flow is perhaps as simple as it can be while still capturing the idea of asynchronous processing. In its simplest form, the flow is as follows:

- The device is seized for exclusive access. Until this `do_io` request is complete, other requests will block at this point.
- The type of operation (*e.g.*, seek, write, rewind) is sent to the device.
- The requested operation is performed by the device. If the operation is an output-type operation (*e.g.*, seek, write), user-supplied data are sent to the device. If it is an input-type operation (*e.g.*, read), data from the device are transferred to a user-supplied buffer area. While the operation is in progress, other Path Pascal processes may proceed; the requesting process blocks at a semaphore until the data transfer is completed.
- The device returns a status describing the success or failure of the operation. This status is returned as a value by `do_io`. Prior to returning, if the system is operating in simulated time, the device simulator performs a `delay` operation so that the I/O terminates at the correct simulated time. Pseudo-real-time I/O always terminates as soon as possible.

In addition to the above sequence, some communication with I/O devices is initiated by the device rathe than the program. Examples of this are the user pressing the BREAK key on a terminal, an operator mounting or dismounting a removable storage medium, and a sudden failure of a device. We shall see further examples of this type of operation in the discussion of the `multiplexer` device type. These unrequested conditions on devices are called "urgent conditions" or "urgent interrupts."

The Path Pascal program wishing to handle an urgent condition on a device may do so by executing that device's `await_int` request. This request, in its simplest form, has the following flow:

- The device is seized for awaiting the interrupt. Until some urgent condition is detected, other `await_int` requests will block at this point. Note that this seizing of the device and that performed by `do_io` are orthogonal; *i.e.*, a `do_io` and an `await_urg` request may be outstanding at the same time.
- The Path Pascal process is blocked until an urgent condition is detected. Other Path Pascal processes may proceed.
- When some urgent condition occurs, the device sends a status indication describing the condition. The Path Pascal process that executed the `await_int` request is unblocked, and the `await_int` function returns the status as its value.

Finally, a `close` operation is provided on the `device` object. This operation indicates that the device is no longer required, terminates the associated server, and releases the memory used for various internal control structures associated with the device.

## 3. Multiplexer channels.

The above device model serves for the simplest cases of I/O, where all data transfers occur in the sequence, "send request; await interrupt; process status return." It has some drawbacks, however, in some of the more sophisticated uses of I/O.

One problem is that a Path Pascal process must be dedicated to each device on which I/O is in progress. In a system such as a terminal controller, where I/O may be pending on a large number of low-volume devices, the overhead of maintaining all these processes may be prohibitive; some means must therefore be provided for the program to wait for an interrupt from any one of a set of devices.

Another problem relates to the construction of peripherals in the real world. Generally speaking, large computers have multiplexed I/O channels (various manufacturers use various names to describe these) which allow a single channel to be used for requests for several peripherals, but impose the restriction that only one may actually transfer data at a time. For instance, it is typical to have many disk drives attached to a single I/O path, and to be able to have all of them seeking at once. Only one at a time, however, may execute read or write operations.

These situations are incorporated in the Path Pascal device model by a special device type, called a "multiplexer channel." Each multiplexer channel represents a single I/O path to multiple devices. The set of I/O requests accepted by the multiplexer channel is the union of the requests accepted by the individual devices attached to it, plus several requests specific to the channel itself; the most important of these is `select`, which transmits a device number to the channel. The device number identifies a device to be used for a subsequent I/O request.

Multiple requests can operate in parallel on a multiplexer channel; nevertheless, the device model presented in Section 2 still applies. When a request is sent to a multiplexer channel, the process that sent it still blocks until the operation is completed. The

apparent contradiction is resolved, and the capability to operate in parallel is gained, by introducing a new set of requests, called "non-blocking operations."

A non-blocking operation is an operation that will take some time following a data transfer before it completes. An example of such an operation is a seek on a disk drive, which transfers the seek address immediately but will typically take many milliseconds before addressing the requested track. (An even more extreme example is a rewind operation on a tape drive, which may take up to several minutes.) On a device which accepts a non-blocking version of one of these operations, as soon as all data transfer is complete, the device returns a status code indicating that an operation is in progress. The multiplexer channel is then freed for other work.

When the actual work for a non-blocking operation is complete, the device generates an urgent condition. The status code presented with the interrupt indicates the success or failure of the original request. The program (which has presumably issued an `await_int` request) receives the status and is free once again to send commands to the device.

Of course, since the device is attached to a multiplexer channel, there is the possibility that a `do_io` request is pending on another device, so a further request for this device may have to wait. The runtime system handles this case automatically, since no device (including a multiplexer channel) may have more than one `do_io` request in operation at a time.

The multiplexer channel device type imposes some additional requirements on the device software in the Path Pascal runtime system. In particular, the `await_int` request handler in the `device` object must execute a `delay` operation when an interrupt is received in simulated-time operation, to bring the simulated time up to the moment when the interrupt occurs.

# Lightweight Processes In An Object–oriented System

**Kevin B. Kenny**

**Department of Computer Science**

**University of Illinois at Urbana–Champaign**

**Urbana, Illinois**

# Appendix E.
# Lightweight processes in an object--oriented system.

*by*

Kevin B. Kenny

## 1. Introduction.

Research is now in progress on defining an operating system which uses an object-oriented programming style even at its lowest levels. As a preliminary study, a dispatcher has been implemented which gives the user the ability to have a number of lightweight processes running in a single domain. A version has been implemented which simulates the operation of a system by time-slicing within a single process under Unix on a VAX. Implementation work is in progress for an actual kernel running on one of the microcomputers in our laboratory.

A number of fairly natural abstract data types were described in the course of this investigation. The types will be presented below, roughly in order by the level of abstraction; primitive types will be presented first.

## 2. Piles.

The implementation of the dispatcher required stacks, queues, and priority queues of several different types of object. Since it was foreseen that these data structures will be required in many other applications, they have been fully generalized with a construct called a `pile`. A `pile` is an object with three fundamental operations:

- Clear the pile.
- Add an object to the pile.
- Locate the "first" object on the pile, and remove it.

The C++ definition of a `pile` is shown in Figure 2.1.

```
class pile {
public:
        virtual pile& operator << (void *);     // Add an item
        virtual void * next ();                 // Remove & return an item
        virtual pile& clear ();                 // Empty the pile
        pile& operator = (pile&);               // Assignment among piles
        pile (pile&);                           // Copy in initialization
        pile () {}                              // Constructor
        };
```

**Figure 2.1. The `pile` data type.**

The research presented in this paper was supported in part by the National Aeronautics and Space Administration under Grant NASA-NSG-1471.

The virtual functions shown in the declaration are not implemented for the `pile` class; instead, they are all replaced by stubs which call a function named `empty_container`. When `empty_container` is called, the program aborts; in the simulation environment on the VAX, an error message is printed indicating that an unimplemented virtual function was called. The virtual functions are replaced with actual ones in the implementation of queue, priority queue, and stack data types.

### 3. Stacks, queues, and priority queues.

With the unifying construct of a `pile` in hand, we now can proceed to build classes for the abstract types `stack`, `queue`, and `pqueue` (priority queue). These are all derived from the `pile` class by adding internal structures to represent the data, and replacing the virtual functions with ones that perform the requisite manipulations. As an example, the declarations for `queue` and `pqueue` are shown in Figure 3.2.

In this implementation (note that the user of `queue` need not be aware of it), queues are maintained by maintaining a linked list of linkage nodes (called `queue_entry`'s). Each of these comprises a pointer to the object in the queue, and a pointer to the next linkage node in the queue. The last linkage node points back to the first, eliminating the need for separate head and tail pointers.

Priority queues are (in this primitive implementation; more sophisticated procedures are expected) implemented as linear lists as well. A priority queue, however, also requires a `comparator` function, which tells the relative priority of two elements; this function is supplied to the priority queue's constructor. A comparator is invoked with a call of the form:

```
(*c) (entry1, entry2);
```

and is expected to return a negative value if `entry1` precedes `entry2`, zero if `entry1` has equal priority to `entry2`, and a positive value if `entry2` precedes `entry1`.

### 4. Generic piles.

As defined, `pile`'s are not very useful; the entries that are added to and removed from them are just void pointers. A set of macros are provided to generalize these to the notion of "generic piles", which can contain arbitrary data. A pile of objects of type `t` is declared with the macro invocation `gpile(t)`; in order to have this class available, the macro invocation

```
declare(gpile, t)
```

must appear in the source file among the other class definitions.

The constructor for a generic pile accepts a reference parameter, which is the pile to be made generic. For example, the dispatcher's ready queue is a pile of `threads` (we shall define these later) organized as a priority queue with comparator function `rq_comp`. The combination is declared with:

```
declare(gpile,thread)
pqueue rq ((comparator) rq_comp);
gpile(thread) ready_queue (&rq);
```

### 5. Rationale: Why generic piles?

At first glance, it may appear that the definition of generic piles is cumbersome; it is not clear why all the complexity is required. The conventional way to program such things would be to code a few subroutines that made insertions and deletions, and to call them directly. The object-oriented approach presented here, though, offers a measure of uniformity and flexibility.

The uniformity comes from the fact that a generic object can be used to represent any type; a queue of processes has the same data structure as a queue of buffers or a queue of messages. The underlying procedures need be implemented and debugged only once, an expected saving of development and maintenance expense.

Furthermore, there is a uniformity of interface. Adding to a priority queue, a stack, or a heap is the same process as adding to a queue; in fact, the routine doing the adding need not even know which type of

```
typedef int (*comparator) (void*, void*);

class queue_entry {
        friend class queue;
        friend class pqueue;

        queue_entry * qe_next;
        void * qe_item;

        queue_entry (void * i, queue_entry * n = 0);
        ~queue_entry ();
        };

class queue : public pile {
        friend class pqueue;

        queue_entry * q_last;
public:
        pile& operator << (void *);             // Add an item
        void * next ();                         // Remove and return an item
        pile& clear ();                         // Clear the queue
        queue& operator = (queue&);             // Assignment operator
        queue ();                               // Constructor
        queue (queue&);                         // Copy in assignment
        ~queue ();                              // Destructor
        };

class pqueue : public queue {
        comparator c;                           // Comparison function
public:
        pile& operator << (void *);             // Add to queue
        pqueue (comparator);                    // Define queue
        pqueue& operator = (pqueue & q) {       // Assignment
                queue::operator = (q);
                return *this;
                }
        };
```

**Figure 3.2. The queue and pqueue classes.**

pile is being manipulated. This uniformity has already been observed to simplify the construction of the dispatcher; the same code adds processes to the ready queue, to the delay queue, and to semaphore queues, despite the fact that these three types of queue have different organizations.

The flexibility comes from the fact that the data structures implementing the types are hidden from the caller. If an implementor has decided to replace (say) the linear list in the priority queue class with a data structure that can be searched in less time, it can be done by changing that class's private data and revising its entry functions. No change (other than recompilation) to the users of the class is needed.

## 6. Machine conditions.

We come now to another primitive data type: machine conditions. This type represents whatever information is necessary to save the state of a lightweight process and restore it again. A typical implementation is shown in Figure 6.3.

The constructor for the machine conditions class is responsible for any work needed to make a thread of control ready to run with the specified function as its entry point. On the VAX, this involves allocating stack space and making an initial stack that describes the entry and provides a point for its ultimate return.

The sole operation which may be performed on a set of machine conditions is "dispatch," which takes whatever action is necessary to suspend the dispatcher's flow of control and resume that of the thread. On a VAX, "dispatch" switches stacks to the thread, changes the signal context to the thread (several signals are blocked when the dispatcher is executing), and returns to the thread. A dispatch ends with one of the following:

- • A kernel call (via the function `_disp_end`. The parameters to the kernel call are located by the pointer `argv` returned by the `dispatch` call; `argc` gives the number of parameters. The first parameter gives an ordinal number of the function to be performed.
- An interrupt (on the VAX simulation, a Unix signal). The state of the thread is saved, and `argc` and `argv` are set up as if the thread executed a kernel call with function number `D_INTERRUPT`. The remaining arguments give enough information to locate the source of the interrupt (under Unix,

```
typedef int (*entryfunct) (...);

class Machine_Cond {
        void *frame_pointer;            // Frame pointer for next dispatch
        void *stack_base;               // Base of the stack area
        long num_entries;               // Size of the stack area in longwords
public:
        void dispatch (int &argc, int *&argv);
                                        // Procedure to dispatch to the
                                        // specified machine conditions.
                                        // Returns argument list from the
                                        // first kernel call performed.
        Machine_Cond (entryfunct entrypoint,
                      int argc = 0,
                      void * argv = 0,
                      int stack_size = 0);
                                        // Constructor takes entry address,
                                        //      argument list description, and
                                        //      (optional) stack space needed.
        Machine_Cond ();                // Default initializer
        Machine_Cond (Machine_Cond&);   // Copy in initialization
        void operator= (Machine_Cond&); // Copy in assignment
        ~Machine_Cond ();               // Destructor is needed.
        };
```

**Figure 6.3. "Machine conditions" class.**

the signal number and signal code are passed).

- The entry function of the thread returns. `argc` and `argv` are set up as if the thread executed a kernel call with function number `D_END`. A second argument is supplied, which gives the return value from the entry function.

In order to perform the hardware support required, a fairly extensive amount of assembly language programming was done; the compiler has no primitives to support multiple threads of execution.

## 7. Threads.

We finally come to the `thread` data type, which is the fundamental data type describing a lightweight process. It contains the process's machine conditions, a "wait-for-son" semaphore (which will be described later), an indicator of the thread's state (ready, running, or blocked), the thread's priority, the thread's wakeup time if the thread is delayed, and the thread's termination status if it has terminated. It has entry functions that make the thread ready to run (by placing it on the ready queue) and dispatch to the thread. Its declaration is shown in Figure 7.4.

---

**Figure 7.4. The thread data type.**

```
typedef enum {                          // State of a thread
        T_READY,                                // On the ready queue
        T_RUNNING,                              // In execution
        T_BLOCKED,                              // Blocked
        T_DEAD,                                 // Terminated
        T_BURIED                                // Terminated and destroyed
        } thread_state;

class thread {
        thread * next_thread;           // Next thread on the list of all
                                        //  threads.
        Machine_Cond mc;                // Machine conditions
        semaphore wfs;                  // Wait-for-son semaphore
public:
        thread_state state;             // Current state of the thread
        long vcputime;                  // Virtual CPU time consumed so far
        int priority;                   // Current priority
        int qpriority;                  // Copy of current priority. When a
                                        //      task is on the ready queue,
                                        //      priority may be changed;
                                        //      the ready queue is therefore
                                        //      sorted by qpriority so that
                                        //      it is never out of sequence.
        long wakeup_time;               // Time to wake up if task is delayed.
        int termination_status;         // Status with which a DEAD thread
                                        //      died.

        void ready();                   // Place task on ready queue.
        void disp_proc();               // Dispatch task and process kernel
                                        //      service request.

        thread (entryfunct, int, void*, int=0, int=0);
                                        // Constructor accepts entry function,
```

```
                                        //      argument count,
                                        //      argument list pointer,
                                        //      thread priority,
                                        // and  stack size.

    thread (thread&);                   // Copy in initialization.
    thread& operator = (thread&);       // Assignment among threads.
    ~thread ();                         // Destructor does wait-for-sons.
    };
```

**Figure 7.4 (continued). The thread data type.**

---

The constructor of a thread initializes the local data to the thread (the machine conditions, the priority, and the virtual CPU time), and links the thread onto a list of all threads known to the system. This list has no use in the present implementation, but is provided so that such features as deadline scheduling and dynamic adjustment of priorities may be provided later.

The constructor then links the thread on the ready queue so that it may be executed independently of the parent process. Thereafter, both threads execute under control of the dispatcher.

While a thread is in execution, it may refer to data located in its parent's activation record (for instance, the parameters passed to it may be located there). Because of this, the parent's activation record may not be destroyed until the child thread has terminated. One possible approach to ensure that the activation record is still available (used in Ada) is to maintain a usage count for every activation record and free the activation record only when no procedure is using it. This technique was rejected for use with the C++ system, as it necessitates an expensive allocation call on procedure invocations. Moreover, the thread system was designed to work with the existing procedure call/return mechanism of C++, where no such interface is provided.

Instead of the dynamic allocation scheme, instead, a simple wait-for-sons synchronization like that used in Path Pascal [Kolstad; Grunwald] has been implemented. It is interesting to note that no changes to the language call/exit mechanism were needed to accommodate this implementation; the wait-for-son is accomplished by having a semaphore (called `wfs`) as a component of the `thread` object. When the activation record containing a thread is to be destroyed, the thread's destructor is called. Within the destructor (in addition to operations for freeing the process stack and machine conditions) there is a P operation on this semaphore, which has the effect of waiting until the corresponding V is issued at thread termination time.

Having the wait-for-son synchronization has also proven valuable in doing a generalized synchronization operation; if a set of tasks must be performed in no specific order, with their parent waiting until all have completed, this may be done by spawning a thread for each task and having the successive wait-for-son operations delay the parent's termination until all the children have finished.

The thread's operation is controlled by the system dispatcher making calls to the `dispatch` entry function. This function marks the thread as "running," and calls the "dispatch" entry of its machine conditions object to dispatch to it. It records the virtual CPU time used, and then selects one of the following actions based on the way dispatch was ended.

- If dispatch ended because the thread ended, the thread is marked "dead" and the return value of the entry function is recorded as its termination status. A V operation is executed on the wait-for-son semaphore to re-awaken the parent thread if it blocked waiting for this thread to terminate.
- If dispatch ended because of an interrupt, the thread is returned to the ready queue, and the function, `catch_interrupt`, is called to field the interrupt exactly as if it had occurred in the kernel.

- If dispatch ended because the thread asked to be blocked on some queue, the thread object is added to that queue and its state is changed to "blocked."

Because of the flexibility of the "pile" construct, these three functions have proved sufficient thus far to handle all the dispatching requirements of the system.

## 8. Semaphores.

In the previous section, the wait-for-sons operation was described in terms of semaphore operations. As the reader might have already surmised, basic synchronization is implemented with semaphores. [Dijkstra]

The semaphore construct was chosen because it is easy to implement and is a fundamental component of more sophisticated synchronization primitives. A major intent of this research is to explore synchronization issues in an object-oriented multiprogramming system; a full discussion of these is beyond the scope of this paper.

A semaphore is an object which comprises a counter and a list of processes blocked at the semaphore. It accepts two fundamental operations, P and V, and may also be read as an integer, in which case the value of the counter is returned. The definition of a semaphore is shown in Figure 8.5; note that the P and V operations are implemented as `friend` functions. The choice of `friend` functions rather than entry points was done for notational convenience; it is more familiar to write `P(s)` and `V(s)` rather than the more obscure, if more technically correct, `s.P()` and `s.V()`.

A P operation decrements the semaphore counter. If the count goes negative, the process which executed the P operation is blocked and joins the queue of processes at the semaphore.

A V operation increments the semaphore counter. If the count is non-positive, the process which did the least recent P operation is removed from the queue of processes at the semaphore and made ready (by calling the `ready` entry point of the `thread` object representing it).

```
class semaphore {
        friend semaphore& P(semaphore&);
        friend semaphore& V(semaphore&);
        int count;                      // Semaphore counter.
        queue q;
        gpile(thread) *waiting;         // Queue of threads blocked at
                                        //      semaphore
public:
        operator int () { return count; }       // Examine count
        semaphore& operator = (semaphore&);     // Copy semaphores
        semaphore (semaphore&);
        semaphore (int);                // Initialize semaphore
        semaphore ();
        ~semaphore ();                  // Destroy semaphore
        };

semaphore& P (semaphore& s);
semaphore& V (semaphore& s);
```

**Figure 8.5. The semaphore data type.**

Both of these operations need some special support to make sure that they take place atomically. On a system with only one processor, this support can consist simply of masking interrupts while the operation is being performed. On a system with multiple processors sharing memory, some sort of hardware synchronization, such as a test-and-set loop is required. On the VAX simulation system, the atomicity is guaranteed by masking signals while the operation is in progress; if a P operation blocks, the signal mask is reset by the dispatcher call. For this reason, the design decision was made to save the signal mask of a thread between dispatches only if the thread left execution as the result of an interrupt.

## 9. Interrupt handling.

Having semaphores, and having the capability to interrupt a process in execution, we now can define the discipline by which the system handles interrupts; *i.e.*, conditions which can pre-empt the execution of a process.

We assume that any interrupt in which the system is interested has an interrupt process corresponding to it. This process is responsible for performing an endless cycle of servicing one interrupt and waiting for another. The waiting is accomplished by a P operation on a semaphore corresponding to the type of interrupt; thus the code for a typical interrupt process would be as shown in Figure 9.6.

The `register_interrupt` call is used to inform the interrupt handler that the system expects to handle a particular interrupt. It adjusts the interrupt masks so that the interrupt will be handled when a thread is executing (barring its being explicitly blocked, for example in the P operation of a semaphore). It also keeps the interrupt from occurring in the dispatcher (again by setting interrupt masks). It initializes the `sigsem` semaphore corresponding to the interrupt to allow synchronization of the interrupt process. It may also need to perform hardware-related initialization, *e.g.*, sending data to a priority interrupt controller.

When an interrupt occurs, there are two possibilities: either a thread was in execution, or the processor was idle (Interrupts are inhibited in the dispatcher). If a thread was in execution, its state is saved, and the dispatch ends (at the machine condition level). The `dispatch` routine in the `thread` object gets arguments indicating that an interrupt has occurred and giving its type; it returns the interrupted thread to the ready queue and calls `catch_interrupt` to process the interrupt.

```
extern semaphore *sigsem [];            // Vector of semaphores corresponding
                                        // to the interrupts to be processed.

int interrupt_process (...) {
        //
        // Do whatever initialization is necessary.
        //
        register_interrupt (number)     // Tell the interrupt catcher that we
                                        // want to handle an interrupt.
        for (;;) {
                P(*sigsem [number]);    // Wait for the interrupt
                // Process the interrupt.
                };
        }
```

**Figure 9.6. Typical interrupt process.**

If an interrupt occurs while the processor is idle, no special processing is needed for thread control, and the interrupt handler calls `catch_interrupt` directly.

Either way, control arrives at `catch_interrupt` with the dispatcher in control of the processor and the interrupt information passed as parameters. The information is used to deduce an interrupt number (on the VAX, this is simple — the interrupt number is simply the Unix signal number) and a V operation is executed on the corresponding semaphore. Control returns to the dispatcher.

In most cases, the interrupt thread will have been blocked on the semaphore, and the V will have returned it to the ready queue. If its priority is higher than that of the interrupted thread (as it usually will be), then it will be dispatched before the interrupted thread, a rudimentary form of pre-emptive scheduling. It also may change the priorities of threads, including ones on the ready queue; these changes will take effect on the next dispatches to the threads.

## 10. Timers.

One particularly important case of an interrupt handler is the thread that manages the interval timer; this timer is used to perform time-slicing among threads and to schedule wakeups. Since the simulated system on the VAX was designed to run in either real or simulated time, the decision was made to make timer management intimate with the dispatcher.

The user interface to the timer management comprises several routines:

`long cpuclock ()`
: This routine returns the amount of virtual CPU time consumed by the entire system so far. Successive values of it are subtracted to give the amount of CPU time to charge to a process for one dispatch.

`long wallclock ()`
: This routine returns the current time, in milliseconds past some time in the past. It is used when the system is in real-time mode to determine the wakeup time for a `delay` operation.

`long clock ()`
: This routine returns the simulated clock. In simulated time mode it is the amount of virtual time consumed by `delay` and `await` operations; in real time mode it is synonymous with `wallclock`.

`void await (long)`
: This routine waits until a particular absolute time, specified in milliseconds. Its primary use is in a cyclic task which wants to begin at particular intervals of time: such a task might appear as shown in Figure 10.7.

`void delay(long)`
: This entry delays the executing thread for a specified amount of time. It is useful in cases where the thread wants to be reawakened periodically; it is also used in simulation studies to simulate the

```
long time = clock();                    // Record time first cycle began.
do {                                    // Cycle...
        // ... Do whatever is required.
        time += interval;               // Compute the next time to run.
        await (time);                   // Wait until that time
        } while (time < stop_time);     // Quit when time has elapsed.
```

**Figure 10.7. Typical task requiring cyclic scheduling.**

duration of events. The two forms, `delay(n)` and `await(n+clock())`, are semantically identical.

All the procedures except `await` (and hence `delay`) are effectively passive; they just read the values of clocks. The active `await` procedure interacts with the thread managing the interval timer as follows. Note that it and the clock process require full mutual exclusion, which is enforced by the use of a binary semaphore.

- If there is an interval timer already set, and it is due to ring later than the wakeup time requested, return the thread which requested it to the delay queue.
- If we are executing in real time, perform a V operation on the interval timer's semaphore to simulate a wakeup and allow the clock process to check the delay queue.
- Enqueue this process on the delay queue. (Some special machinations are required here to make sure that the enqueue happens atomically with releasing the mutual exclusion and allowing the clock thread to execute).

There are two kinds of clock thread, corresponding to simulated time and real time. In simulated time, the clock thread is always ready, with the lowest priority of any thread in the system. When it is dispatched (and, hence, no other thread is ready), it advances the simulated time to the time that the next thread on the delay queue is due to wake up, and awakens it; it then returns itself to the ready queue (behind the just-awakened thread) and allows execution to proceed. If the delay queue is empty, it halts awaiting an interrupt (the use of interrupts in simulated-time mode is not recommended).

In real time, the clock thread runs with a very high priority (so that it can pre-empt other threads). It examines the first thread on the delay queue. If it is due to wake up, it readies the thread and examines another. If it is not due, it sets the interval timer to generate an interrupt at the thread's wakeup time, and awaits another timer interrupt.

Verification that the two clock threads just described (for the actual C++ code, see one of the Appendices) actually interact correctly to implement the `await` function is left as an exercise for the reader.

## 11. The main dispatcher.

With all of the preceding structures defined, the main function of the dispatcher is trivially simple:

- Construct threads for the appropriate clock and for the user's main program.
- While the main program has not terminated, pull threads from the ready queue and dispatch to them. If the ready queue is empty and the main program still has not terminated, delay (*i.e.*, idle the processor) until an interrupt occurs, and repeat the check on the ready queue.

Listings of all the procedures to implement the objects described here are presented in an Appendix.

## 12. Future plans for the system.

Work is in progress on extending the system described here in several ways:

- Models of concurrency other than shared memory (*e.g.*, message queues and remote procedure calls) are being implemented.
- The extension of this kernel to provide other functions is being contemplated; the most likely next step is the implementation of a set of device drivers as active objects (i.e., objects which own their own threads of control).
- The possibility of implementing a kernel for a Unix-like system with this set of procedures as its dispatcher is being investigated. In order to do this, a signal mechanism must be implemented; beyond this nothing else appears to be required in the way of dispatching and synchronization provided that a dispatch always ends with the processor in supervisor state.
- Extending the synchronization primitives to a richer set than simple semaphores is also being investigated. A particularly interesting line of research is to investigate how well such constructs as mediators, [Grass] monitors, [Hoare] guarded commands, and Ada rendezvous will map to an object-oriented

system. In several cases it appears that it will be possible to construct objects that implement these constructs in a fairly automatic manner, without requiring any extensions to the base language; it is an intriguing possibility that C++, without originally having been designed for multiprogramming, may nevertheless accept such constructs gracefully.

```
/*
 * $Header: dispatch.h,v 1.1 86/10/07 14:57:40 kenny Exp $
 *
 *      Dispatcher functions for NULLIX kernel.
 *
 * $Log:        dispatch.h,v $
 * Revision 1.1  86/10/07  14:57:40  kenny
 * Initial revision
 *
 */

typedef enum {
        D_UNKNOWN,              // Unknown operation
        D_END,                  // Terminate a thread
        D_INTERRUPT,            // Accept an interrupt
        D_ENQ                   // Block a thread on a queue
        } kernel_funct;
```



```
/*
 * $Header: empty.c,v 1.1 86/09/30 17:51:37 kenny Exp $
 *
 *      Stub routine for empty container types.
 *
 *      Certain class types in the NULLIX kernel, e.g., pile, are `container'
 *      types; it is not intended that they ever be instantiated themselves,
 *      rather, derived types from them will be created.  If a virtual
 *      function belonging to one of these container types is ever called,
 *      control comes to ``empty_container'' to process the problem.
 *
 * $Log:        empty.c,v $
 * Revision 1.1  86/09/30  17:51:37  kenny
 * Initial revision
 *
 */

#include <stream.h>

extern void abort();

void empty_container (char * entryname) {

        cerr << "Call to unspecified virtual function " << entryname
                << "; cannot proceed.\n";
        cout.flush ();
        cerr.flush ();
        abort ();
        }
```

```
/*
 * $Header: gpile.h,v 1.1 86/09/30 23:50:19 kenny Exp $
 *
 *      Header file for the ``generic pile'' construct.
 *
 * This header file defines macros for implementing a ``pile'' (see pile.h)
 * of objects of any type.  The three macros are:
 *
 *      declare(gpile,type)             // Declares the ``gpile(type)'' class
 *      implement(gpile,type)           // Implements the ``gpile(type)''
 *                                      // class; may be used only once per
 *                                      // type per linked program.
 *      gpile(type)                     // Specifies that an object is to be
 *                                      // a generic pile of objects of a
 *                                      // specified type.
 *
 * Example:
 *      The ready queue is a pile of threads, organized as a priority queue
 *      with the comparison function being ``prio_d'' (descending ordering
 *      by priority).  The declaration of this structure is:
 *
 *      declare(gpile,thread);
 *      implement(gpile,thread);
 *      ...
 *      pqueue readyq_ ((comparator) prio_d);
 *      pile(thread) readyq(reqdyq_);
 *
 * $Log:        gpile.h,v $
 * Revision 1.1  86/09/30  23:50:19  kenny
 * Initial revision
 *
 */

#ifndef GPILEH
#ifndef GENERICH
#include <generic.h>
#endif GENERICH
#ifndef PILEH
#include "pile.h"
#endif PILEH

#define gpile(type) name2(gpile,type)

#define gpiledeclare(type)                                              \
class gpile(type) {                                                     \
        pile *p;                                                        \
public:                                                                 \
        gpile(type)& operator << (type * x) {                           \
                *p << (void *) x;                                       \
                return *this;                                           \
                }                                                       \
        type * next () {                                                \
                return (type *) p -> next ();                           \
                }                                                       \
        gpile(type)& clear () {                                         \
                p -> clear ();                                          \
```



```
                return *this;                                           \
                }                                                       \
        gpile(type) (pile *q) {                                         \
                p = q;                                                  \
                }                                                       \
        gpile(type)& operator = (gpile(type)& x) {                      \
                *p = *(x.p);                                            \
                return *this;                                           \
                }                                                       \
        gpile(type) (gpile(type)&);                                     \
        gpile(type) ();                                                 \
        };

#define gpileimplement(type)                                            \
gpile(type)::gpile(type) (gpile(type)& x) {                             \
        void empty_container (char *);                                  \
        gpile(type) *foo = &x;                                          \
        empty_container (                                               \
                "gpil\145(type)::gpil\145(type)"),                      \
        }                                                               \
gpile(type)::gpile(type) () {                                           \
        void empty_container (char *);                                  \
        empty_container ("gpil\145(type)::gpil\145(type)");              \
        }

#define GPILEH
#endif GPILEH
```

```
/*
 * $Header: hardware.c,v 1.4 86/10/08 00:51:25 kenny Exp $
 *
 *      Hardware-dependent routines for C++ thread package.
 *              (VAX version)
 *
 * $Log:        hardware.c,v $
 * Revision 1.4  86/10/08  00:51:25  kenny
 * Made change to allow creating threads with default stack size.
 *
 * Revision 1.3  86/10/07  14:58:45  kenny
 * Added changes to allow signal processing to use a variable kernel mask and
 * added additional code for interrupt handling.
 *
 * Revision 1.2  86/09/24  19:54:08  kenny
 * Added changes to make sure that machine conditions aren't copied (which
 * would be disastrous as it would get two processes in the same stack).
 *
 * Revision 1.1  86/09/23  12:45:28  kenny
 * Initial revision
 *
 *
 */

#include "hardware.h"
#include <stream.h>

static char RCSID [] = "$Header: hardware.c,v 1.4 86/10/08 00:51:25 kenny Exp $";

/*      Machine condition constructor   */

/*      The constructor for a machine condition accepts an entry point and
        a stack size (with zero indicating the default of using
        Default_Stack_Size).  It constructs a set of machine conditions that
        will take execution to the specified entry point on thread start, and
        to the dispatcher ``exit'' function on thread termination.  These
        conditions look as follows:

                                                Stack area (low addresses
                        MachineCond                     toward top of page)
                        +---------------+       +---------------+
        stack_base      |       :-------+------>= stack growth  =
                        +---------------+       +---------------+
        frame_pointer   |       :-------+------>A 0             |
                        +---------------+       | psw           |
                                                | (ap)  :-------+----.
        entrypoint:     .word 0xXXXX            | (fp)  :-------+--. |
                        (first instr.)  <-------+-(pc)          |  | |
                                                + - - - - - - - +  | |
                                                B 0             |<-' |
                                                | psw           |    |
          ``thread end''  .word 0x0000          | (ap) :--------+----+--.
                        movl    4(ap), r0       | (fp) :--------+----+--. |
                        movl    r0, -(sp) <-----+-(pc)          |    |  | |
                        movl    $1, -(sp)       + - - - - - - - +    |  | |
                        calls   $2, __disp_end  | argc          |<---'  | |
                                                | argv          |       | |
                                                =  ...          =       | |
                                                + - - - - - - - +       | |
                                                C 0             |<------' |
                                                | psw           |         |
                                                | (ap)  -1      |         |
                                                | (fp)  -1      |         |
                                                | (pc)  -1      |         |
                                                + - - - - - - - +         |
                                                | 0             |<--------'
                                                +---------------+
                        halt
```



```
        The interpretation of this setup is as follows:

        When the first dispatch occurs, the innermost frame (A in the
        diagram) is pulled off the stack, and its ap and fp are placed in
        the machine registers.  Control transfers to the entry point of
        the main procedure of the thread, exactly as if a call had been done
        from the point described by frame B.  The apparent arguments to the
        procedure are described by argc and argv as passed to the constructor.

        When the main procedure of the thread returns, control transfers to
        the state described in frame B.  Ap and fp are loaded from the frame
        (designating dummy frame C) and control passes to the ``thread end''
        procedure, which does an ``end of dispatch'' call, function 1, to
        take the thread out of execution.  The return value (in register 0)
        is passed to the dispatcher to be reported to the static parent of the
        thread.

        Frame C is a dummy frame so that ap and fp always have ``meaningful''
        values.  It is never the current frame, and will cause a machine
        fault is a dispatcher bug transfers control to it.
*/

int Default_Stack_Size = DEFAULT_STACK_SIZE;

Machine_Cond::Machine_Cond (    entryfunct entrypoint,
                                int argc,
                                void * argv,
                                int stack_size) {

        extern void * _thread_retaddr;  // Address in assembly code to which
                                        // the main procedure of a thread
                                        // returns.

        union stack_entry {             // Format of a stack entry
                long *pl,
                void *pv,
                stack_entry *ps,
                long l,
                },

        // Allocate space for the stack, and find its top

        if (stack_size == 0) stack_size = Default_Stack_Size;
        num_entries = (stack_size + sizeof (stack_entry) - 1)
```

```
                        / sizeof (stack_entry);
        stack_entry * stack = (stack_entry *) new long [num_entries];
        stack_base = (void *) stack;
        stack_entry * stack_ptr = stack + num_entries;

        // Fill in the stack frames

        (--stack_ptr) -> l = 0L;        // Null arglist for 'B' frame
        stack_entry * b_arglist = stack_ptr;

        (--stack_ptr) -> l = -1L;       // Dummy PC in 'C' frame
        (--stack_ptr) -> l = -1L;       // Dummy FP
        (--stack_ptr) -> l = -1L;       // Dummy AP
        (--stack_ptr) -> l = 0L;        // PSW
        (--stack_ptr) -> l = 0L;        // Condition handler
        stack_entry * c_frame = stack_ptr;

        [ for (register int n = argc - 1; n >= 0; --n)
                (--stack_ptr) -> l = ((long *)argv) [n]; ]
                                        // Argument list
        (--stack_ptr) -> l = argc;      // Argument count
        stack_entry * a_arglist = stack_ptr;

        (--stack_ptr) -> pv = _thread_retaddr;  // PC in 'B' frame
        (--stack_ptr) -> ps = c_frame;          // FP
        (--stack_ptr) -> ps = b_arglist;        // AP
        (--stack_ptr) -> l = 0L;                // PSW
        (--stack_ptr) -> l = 0L;                // Condition handler
        stack_entry * b_frame = stack_ptr;

        (--stack_ptr) -> pv = (void *) ((char *) entrypoint + 2);
                                        // PC in 'A' frame
        (--stack_ptr) -> ps = b_frame;          // FP
        (--stack_ptr) -> ps = a_arglist;        // AP
        (--stack_ptr) -> l = 0L;                // PSW
        (--stack_ptr) -> l = 0L;                // Condition handler

        // Save the current top of stack in machine conditions

        frame_pointer = (void *) stack_ptr;
        }

/*      Copy machine conditions in initialization       */

Machine_Cond::Machine_Cond (Machine_Cond& copy) {
        void empty_container (char*);
        Machine_Cond *c=&copy;          // Shut up the compiler
        empty_container ("Machine_Cond::Machine_Cond (Machine_Cond&)");
        }

/*      Copy machine conditions in assignment   */

void Machine_Cond::operator= (Machine_Cond& copy) {
        void empty_container (char*);
        Machine_Cond *c=&copy;          // Shut up the compiler
        empty_container ("Machine_Cond::operator=");
```



```
        }

/*      Default initializer     */

Machine_Cond::Machine_Cond () {
        void empty_container (char*);
        empty_container ("Machine_Cond::Machine_Cond ()");
        }

/*      Destructor for machine conditions -- just free up the stack     */

Machine_Cond::~Machine_Cond () {
        delete [num_entries] (long *) stack_base;
        }
```

```
/* $Header: hardware.h,v 1.6 86/10/07 14:59:52 kenny Exp $
 *
 *      Hardware-dependent definitions for the C++ library
 *
 * $Log:        hardware.h,v $
 * Revision 1.6  86/10/07  14:59:52  kenny
 * Added changes to allow variable interrupt mask in kernel as well as user.
 *
 * Revision 1.5  86/10/01  15:01:28  kenny
 * Changed declaration of _catch_signal to be compatible with fixed version
 * of <signal.h> (Thanks, Bob!).
 *
 * Revision 1.4  86/09/25  14:18:50  kenny
 * Changed ``entryfunc'' declaration to variable argument list to avoid type
 * incompatibilities.
 *
 * Revision 1.3  86/09/24  19:54:56  kenny
 * Added changes to make sure that machine conditions aren't copied (which
 * would be disastrous as it would get two processes in the same stack).
 *
 * Revision 1.2  86/09/23  20:11:40  kenny
 *
 *
 * Revision 1.1  86/09/23  12:46:25  kenny
 * Initial revision
 *
 */

#ifndef HARDWAREH
static const int DEFAULT_STACK_SIZE = 2032;
                                // Default stack space for a thread

extern int Default_Stack_Size;

typedef int (*entryfunct) (...);

/* Description of the 'machine condition' structure in the thread control
        block */

class Machine_Cond {
        void *frame_pointer;            // Frame pointer for next dispatch
        void *stack_base;               // Base of the stack area
        long num_entries;               // Size of the stack area in longwords
public:
        void dispatch (int &argc, int *&argv);
                                        // Procedure to dispatch to the
                                        // specified machine conditions.
                                        // Returns argument list from the
                                        // first kernel call performed.
        Machine_Cond (entryfunct entrypoint,
                      int argc = 0,
                      void * argv = 0,
                      int stack_size = 0);
                                        // Constructor takes entry address and
                                        //      (optional) stack space needed.
        Machine_Cond (),                // Default initializer
```



```
        Machine_Cond (Machine_Cond&);   // Copy in initialization
        void operator= (Machine_Cond&); // Copy in assignment
        ~Machine_Cond ();               // Destructor is needed.
        };

void _disp_end (int function, ...);     // End-of-dispatch in task.

int _catch_signal (int, int, struct sigcontext *);      // Signal catcher.

extern int user_mask;                   // Thread's interrupt mask.
extern int kernel_mask;                 // Kernel's interrupt mask.

extern int _disp_context_switches;      // Number of context switches done.
extern int _disp_interrupts;            // Number of interrupts
extern int _disp_ints_in_kernel;        // NUmber of interrupts in kernel.

#define HARDWAREH
#endif HARDWAREH
```

```
/*
 * $Header: hardwarel.s,v 1.3 86/10/07 15:01:00 kenny Exp $
 *
 *      Multi-thread execution for the C++ system - assembly language
 * dispatcher.
 *
 *      This file contains the assembly language components required to
 * dispatch among various threads of execution in a single C++ domain;
 * note that it is (of course) specifically dependent on the VAX architecture.
 * It contains the following entries:
 *
 * void Machine_Cond::dispatch (int &argc, void *&argv);
 *      (Called only by the C-level dispatcher).
 *      Saves the state of the C-level dispatcher.  Restores the context of
 *      the thread of execution described by the machine conditions.  The
 *      external variable, this_thread, is set to the current machine
 *      condition  pointer.
 *
 * void _disp_end (int function, ...);
 *      (Called only by runtime support routines for the tasking system)
 *      Saves the state of the currently executing thread and restores the
 *      context of the dispatcher.  The argc and argv supplied as reference
 *      parameters on the call to dispatch are filled in according to the
 *      arguments supplied to _disp_end.
 *
 * void _thread_end (int status);
 *      (May not be called by a thread explicitly; is implicitly called when
 *      the main program of a thread returns, using the return value as the
 *      status)
 *      Requests that a thread of execution be terminated.  The status
 *      presented is the return value of the main program of the thread.
 *      No wait-for-sons synchronization is done here; rather the destructor
 *      of the task does the wait-for-son.
 *
 * void _catch_signal (int sig_no, int sig_code, struct sigcontext *scp);
 *      (Called by being established as a Unix signal handler via a call
 *      to the ``signal'' routine.)
 *      Handles a signal from one of two places:
 *      1)      At the kernel idle loop.  In this case, the signal is serviced
 *              by calling catch_interrupt (sig_no, code).  The idle loop is
 *              then resumed; the sigpause at which it was interrupted is
 *              broken, causing the dispatcher to search for further activity.
 *      2)      In a thread.  The thread's context is saved and the thread is
 *              taken out of execution as if a _disp_end (2, sig_no, sig_code)
 *              had been executed.
 *
 * $Log:        hardwarel.s,v $
 * Revision 1.3  86/10/07  15:01:00  kenny
 * Made changes to allow for variable interrupt mask in kernel as well as user.
 *
 * Revision 1.2  86/09/23  20:11:48  kenny
 *
 *
# Revision 1.1  86/09/23  12:47:08  kenny
# Initial revision
#
```



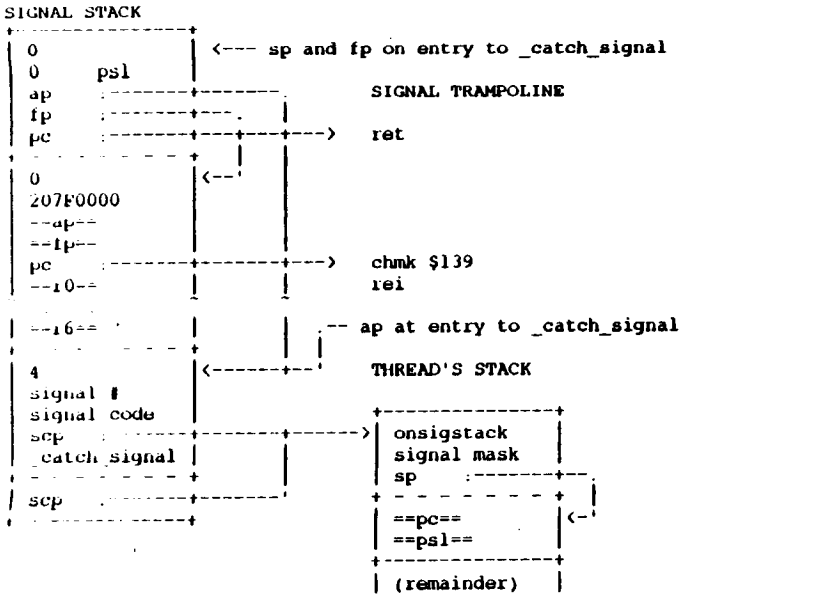
```
 */

        .data
        .asciz  "$Header: hardwarel.s,v 1.3 86/10/07 15:01:00 kenny Exp $"
        .text

/*
 *      Global external variables
 */

        .data
        .align  2
        .globl  __disp_context_switches
__disp_context_switches:
        .long   0               /* Count of dispatches performed */
        .globl  __disp_interrupts
__disp_interrupts:
        .long   0               /* Count of interrupts serviced */
        .globl  __disp_ints_in_kernel
__disp_ints_in_kernel:
        .long   0               /* Count of interrupts from idle routine */

        .globl  _this_thread
_this_thread:
        .long   0               /* Pointer to machine condition of
                                   executing thread */


        .globl  _user_mask
_user_mask:
        .long   0x00000000      /* Interrupt mask for executing thread.
                                        When an interrupt is enabled,
                                        the corresponding bit is reset
                                        here. */
        .globl  _kernel_mask
_kernel_mask:
        .long   0x00000000      /* Interrupt mask for kernel.  When an
                                        interrupt is enabled, the
                                        corresponding bit is set here.
                                        */
        .text

        .globl  __Machine_Cond_dispatch
        .globl  __disp_end
        .globl  __thread_end
        .globl  __catch_signal

#ifdef PROFILE
        .globl  mcount
#endif PROFILE

__Machine_Cond_dispatch:
        .word   0xCFC0                  /* Save r6-r11 for the dispatcher */

#ifdef PROFILE
        moval   L1, r0                  /* Invoke profiler if necessary */
        jsb     mcount
```

```
        .data
        .align  2
L1:     .long   0
        .text
#endif PROFILE

        incl    __disp_context_switches /* Count context switches */
        movl    fp, ___disp_stack       /* Save the dispatcher's stack ptr */
        movl    ap, ___disp_argp        /* Save the dispatcher's arg list */
        movl    4(ap), r8               /* Get pointer to thread's state */
        movl    r8, _this_thread        /* Save pointer to executing thread */
        pushl   (r8)                    /* Switch to thread's signal context*/
        pushl   _user_mask
        clrl    -(sp)
        pushl   sp
        chmk    $139
        movl    sp, fp                  /* Get thread's frame pointer */
        ret                             /* Dispatch to the thread */

__disp_end:
        .word   0xCFC0                  /* Save r6-r11 */

#ifdef PROFILE
        moval   L2, r0                  /* Invoke profiler if necessary */
        jsb     mcount
        .data
        .align  2
L2      .long   0
        .text
#endif PROFILE

        pushl   ___disp_stack           /* Change to kernel signal context */
        pushl   _kernel_mask
        clrl    -(sp)
        pushl   sp
        chmk    $139
        cvtbl   (ap), r7                /* Save argument count */
        moval   4(ap), r8               /* Save argument vector pointer */
        movl    _this_thread, r9        /* Save thread state */
        movl    fp, (r9)
        clrl    _this_thread            /* No thread in execution any more */
        movl    sp, fp                  /* Restore dispatcher state */
        movl    ___disp_argp, ap        /* Get dispatcher argument list */
        movl    r7, *8(ap)              /* Save process argument count */
        movl    r8, *12(ap)             /* Save process argument list ptr */
        ret

        .data
        .globl  __thread_retaddr
        .align  2
__thread_retaddr:
        .long   L3
        .text

__thread_end:
        .word   0x0000                  /* Don't save registers at end */
```

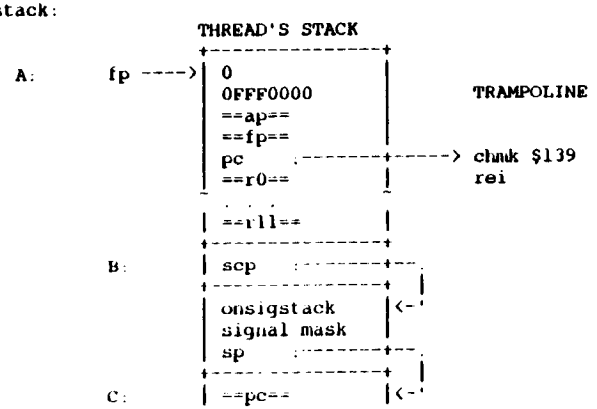


```
        movl    4(ap), r0               /* Get return status for thread */

        /* Normal thread exit starts at next statement */

L3:     movl    r0, -(sp)               /* Save return status for dispatcher*/

#ifdef PROFILE
        moval   L4, r0                  /* Invoke profiler if necessary */
        jsb     mcount
        .data
        .align  2
L4:     .long   0
        .text
#endif PROFILE

        movl    $1, -(sp)               /* Save dispatcher function # 1 */
        calls   $2, __disp_end          /* Go through normal end-of-dispatch*/
        halt                            /* Should never be dispatched again */

/* eject
```

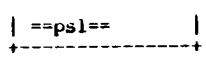
When a signal is caught, the Unix signal mechanism switches to the signal stack automatically. A collection of context is saved on the thread stack and the signal stack for a possible restart:




_catch_signal takes the data thus supplied and builds an additional frame on the thread stack:






Dispatch causes fp to be loaded pointing to the 'A' frame shown. Sp still points to the dispatcher's stack, until the ret is done. Now fp has the thread's frame pointer, ap has its argument pointer, and its arithmetic registers are restored. The sp points at 'B'.

Control transfers to the signal trampoline, which executes a chmk $139. The kernel resets the 'onsigstack' flag, the signal mask, and the stack pointer (all in one atomic operation) and returns with the stack pointer pointing at 'C'.

Finally, the REI instruction in the signal trampoline restores the PC and PSL to their values at the time of the interrupt. SP is now set correctly, also.

```
*/

__catch_signal:
        .word   0x0                     /* Don't save registers on signal */

#ifdef PROFILE
        moval   L5, r0                  /* Invoke profiler if necessary */
        jsb     mcount
        .data
        .align  2
L5:     .long   0
        .text
#endif PROFILE

        /* Handle interrupts in the kernel idle routine */

        movl    _this_thread, r6        /* Get the machine cond. for thread */
        bnequ   L7                      /* If no thread, then we're kernel */

#ifdef PROFILE
        moval   L6, r0                  /* Invoke profiler if necessary */
        jsb     mcount
        .data
        .align  2
L6:     .long   0
        .text
#endif PROFILE

        incl    __disp_ints_in_kernel   /* Count kernel interrupts */
        pushl   8(ap)                   /* Push signal code */
        pushl   4(ap)                   /* Push signal number */
        calls   $2, _catch_interrupt    /* Call kernel routine for int */
        ret                             /* Restart dispatcher */

        /* Handle interrupts from executing thread */

L7:
#ifdef PROFILE
        moval   L8, r0                  /* Invoke profiler if necessary */
        jsb     mcount
```

```
        .data
        .align  2
L8:     .long   0
        .text
#endif PROFILE

        incl    __disp_interrupts       /* Count number of interrupts */
        movl    12(ap),r0               /* Get sigcontext pointer */
        movl    r0, -(r0)               /* Save scp on thread stack */
        movl    r11, -(r0)              /* Save r11-r7 on thread stack */
        movl    r10, -(r0)
        movl    r9, -(r0)
        movl    r8, -(r0)
        movl    r7, -(r0)
        addl3   12(fp), $48, r1         /* Get pointer to r6 in next frame */
        movl    -(r1), -(r0)            /* Save r6-r0 on thread stack */
        movl    -(r1), -(r0)
        movl    -(r1), -(r0)
        movl    -(r1), -(r0)
        movl    -(r1), -(r0)
        movl    -(r1), -(r0)
        movl    -(r1), -(r0)
        movl    -(r1), -(r0)            /* Save trampoline pc */
        movl    -(r1), -(r0)            /* Save ap on thread stack */
        movl    -(r1), -(r0)            /* Save fp on thread stack */
        movl    $0x0FFF0000, -(r0)      /* Make a register save mask */
        clrl    -(r0)                   /* Null condition handler */

        movl    r0, (r6)                /* Update the saved frame pointer */
        clrl    _this_thread            /* We are now executing in kernel */

        movl    8(ap), -(r0)            /* Stack up arguments to dispatcher */
        movl    4(ap), -(r0)
        movl    $2, -(r0)
        movl    ___disp_argp, ap
        movl    $3, *8(ap)              /* 3 arguments */
        movl    r0, *12(ap)

        pushl   ___disp_stack           /* Switch to kernel stack and */
        pushl   _kernel_mask            /*      signal context */
        clrl    -(sp)
        pushl   sp
        chmk    $139                    /* Magic kernel call reloads sp
                                           and changes signal context */
        movl    sp, fp
        ret

/*
 *      local static data
 */

        .globl  __disp_stack            /* Made global for post-mortem */
        .globl  ___disp_argp
        .data
        .align  2
__disp_stack:
```



```
        .long   0
___disp_argp:
        .long   0
```

```
/*
 * $Header: interrupt.c,v 1.2 86/10/08 00:52:44 kenny Exp $
 *
 *      Kernel-level interrupt processing for NULLIX.
 *
 * (Hardware dependent -- UNIX simulation version).  When an interrupt
 * occurs, control transfers automatically to the _catch_interrupt routine in
 * the assembly language kernel.  It switches the stacks so that the system
 * is running in kernel space, and then exits to the dispatcher, which calls
 * catch_interrupt below.  If an interrupt occurs in kernel space (which
 * will happen only when the processor is idle), catch_interrupt is called
 * directly from the assembly language _catch_interrupt routine.
 *
 * Either way, the effect is to V the semaphore corresponding to the interrupt
 * and then return to the dispatcher to get the next ready task, including any
 * task which may have been readied by the V operation.
 *
 * The register_interrupt entry is used to indicate that an interrupt is
 * expected and set the masks accordingly.
 *
 * $Log:        interrupt.c,v $
 * Revision 1.2  86/10/08  00:52:44  kenny
 *   Changed sigsem vector to have the semaphores created on the fly so their
 *   constructors will be called.
 * - Fixed a bug that was improperly clearing bits in the user-level interrupt
 *   mask.
 * - Added a call to sigvec to catch the signal corresponding to any registered
 *   interrupt.
 *
 * Revision 1.1  86/10/07  15:01:49  kenny
 * Initial revision
 *
 */

#include "hardware.h"
#include "semaphore.h"
#include "interrupt.h"
#include <signal.h>

// Unix system interface

int sigsetmask (int);           // Set the interrupt mask
void sigvec (int, struct sigvec_s *, struct sigvec_s *);
                                // Set signal handler

// Vector of semaphores corresponding to the Unix signals

semaphore *sigsem [NSIG];

// Interrupt handling procedure

void catch_interrupt (int signo, int code) {
        register int foo = code,        // Don't tell me about unused args,
                                        // dammit, I know.
        V (*sigsem [signo]),
        }
```



```
// Interrupt registration procedure

void register_interrupt (int signo) {

        struct sigvec_s siginfo;
        struct sigvec_s rubbish;

        register int sm = sigsetmask (kernel_mask);
                                        // Begin critical region
        kernel_mask |= (1 << (signo-1));        // Accept interrupts of this
                                                // type.
        user_mask &= ~(1 << (signo-1)); // Allow this int to user and forbid
                                        // it to kernel.
        sigsem [signo] = new semaphore (0);     // Make semaphore for the
                                        // interrupt.
        (void) sigsetmask (sm);         // End critical section.

        siginfo.sv_handler = (SPF) _catch_signal,
                                        // Set up the signal vector
        siginfo.sv_mask = kernel_mask;
        siginfo.sv_onstack = 1;
        sigvec (signo, &siginfo, &rubbish);
        }
```



ORIGINAL PAGE IS OF POOR QUALITY

```
/*
 * $Header: interrupt.h,v 1.2 86/10/08 00:54:29 kenny Exp $
 *
 *              Interrupt processing routines in NULLIX kernel.
 *
 * (Hardware dependent -- UNIX simulation version).  When an interrupt
 * occurs, control transfers automatically to the _catch_interrupt routine in
 * the assembly language kernel.  It switches the stacks so that the system
 * is running in kernel space, and then exits to the dispatcher, which calls
 * catch_interrupt below.  If an interrupt occurs in kernel space (which
 * will happen only when the processor is idle), catch_interrupt is called
 * directly from the assembly language _catch_interrupt routine.
 *
 * Either way, the effect is to P the semaphore corresponding to the interrupt
 * and then return to the dispatcher to get the next ready task, including any
 * task which may have been readied by the P operation.
 *
 * The register_interrupt entry is used to indicate that an interrupt is
 * expected and set the masks accordingly.
 *
 * $Log:        interrupt.h,v $
 * Revision 1.2  86/10/08  00:54:29  kenny
 * Changed sigsem vector to a vector of pointers to semaphores which will be
 * initialized on the fly (so that the constructors will be called).
 *
 * Revision 1.1  86/10/07  15:02:21  kenny
 * Initial revision
 *
 */

#ifndef INTERRUPTH
#ifndef SEMAPHOREH
class Semaphore;                        // Semaphores exist; all this file
                                        // has to know.
#endif SEMAPHOREH

void catch_interrupt (int, int);        // Process an interrupt (Unix signal)
void register_interrupt (int);          // Set up to catch a particular signal

extern Semaphore *sigsem [];            // Vector of semaphores which are
                                        // automatically P'ed by interrupt
                                        // handler.


#define INTERRUPTH
#endif INTERRUPTH
```



```
/*
 * $Header: main.c,v 1.3 86/10/08 15:41:15 kenny Exp $
 *
 *      Startup for the NULLIX system
 *
 *      This procedure is responsible for getting the NULLIX dispatcher
 * running.  It handles the '-Snn' (user stack space) and '-R' (real-time
 * operations) options from the command line and starts up two threads,
 * ``user_main'' (which is the user-supplied main program) and ``sim_time''
 * or ``real_time'' according to which clock is requested.
 *
 *      It then goes into a loop dispatching to threads from the ready queue,
 * until the user_main thread terminates.  At this point, it shuts the system
 * down.
 *
 * $Log:        main.c,v $
 * Revision 1.3  86/10/08  15:41:15  kenny
 * Fixed bug where user_main stack size was always default, not the size
 * specified on command line.
 *
 * Revision 1.2  86/10/08  00:56:38  kenny
 * - Added code to create a separate stack for catching signals.
 * - Changed code so that the kernel mask will be set correctly, rather than
 *   setting it to -1.
 * - Changed code so that the CPU will go idle only when no task appears on the
 *   ready queue *and* user_main has not terminated.
 * - Added code to pass termination status of user_main back as return status
 *   of the program.
 *
 * Revision 1.1  86/10/07  15:03:12  kenny
 * Initial revision
 *
 */

#include <signal.h>
#include "hardware.h"
#include "gpile.h"
#include "queue.h"
#include "thread.h"

//      Unix system interface

void sigpause (int);                    // Await the occurrence of a signal
int sigsetmask (int);                   // Set the signal mask
void sigstack (struct sigstack_s *, struct sigstack_s *),
                                        // Set the signal stack

int atoi (char *);                      // Convert ASCII to integer

//      Procedures elsewhere in the library

int real_time ();                       // Clock process for real time system
int sim_time ();                        // Clock for simulated time system
int user_main (...);                    // User main procedure

//      Define the comparator that organizes the ready queue
```

```
int rq_comp (void * t1, void * t2) {
        register thread * thread1 = (thread *) t1;      // Promote pointers
        register thread * thread2 = (thread *) t2;
        return thread2 -> qpriority - thread1 -> qpriority;
                                        // Higher priority comes first
        }

//      Define the comparator that organizes the delay queue

int dq_comp (void * t1, void * t2) {
        register thread * thread1 = (thread *) t1;      // Promote pointers
        register thread * thread2 = (thread *) t2;
        return thread1 -> wakeup_time - thread2 -> wakeup_time;
                                                // Nearer time comes first.
        }

//      Initialize all thread lists to ``empty''

thread * first_thread = 0;
thread * current_thread = 0;
qpile(thread) *readyq = 0;
qpile(thread) *delayq = 0;

//      Make the ``simulated time'' flag global.

int sim_time_flag = 1;                  // Default is simulated time.

//      Main program

int main (int argc, char *argv[]) {

        int status;                     // Termination status

{       register char *sigstk = new char [2040];
                                        // Allocate space for signal stack
        struct sigstack_s siginfo;
        struct sigstack_s rubbish;
        siginfo.ss_sp = sigstk+2040;
        siginfo.ss_onstack = 0;
        sigstack (&siginfo, &rubbish);
        }

        int stack = 0;                  // Stack space for user_main

        // Process command line arguments

        char **p = argv+1,
        int i = argc - 1,
        for ( ; i; ++p, --i) {
                                        // Walk through arguments
                if (**p == '-') {       // Look for flags
                        switch ((*p)[1]) {
                                case 'S':
                                        stack = atoi ((*p) + 2);
                                        break;
```



```
                                case 'R':
                                        sim_time_flag = 0;
                                        break;
                                default:
                                        break;
                                }
                        }
                }

        // Initialize the ready and delay queues.

{       pqueue rq ((comparator) rq_comp);
        readyq = new qpile(thread) (&rq);
        pqueue dq ((comparator) dq_comp);
        delayq = new qpile(thread) (&dq);

        // Start up the clock thread

        thread *clock;
        clock = new thread (sim_time_flag ?
                                (entryfunct) sim_time
                              : (entryfunct) real_time, // Entry point
                                0, (void *) 0,          // Arg count and list
                                32767);         // Priority

        // Start up the main thread

        thread *mainpgm = new thread ((entryfunct) user_main,   // Entry point
                                        3, &argc,       // Arg count and list
                                        0,              // Priority
                                        stack);         // Stack size

        // Dispatch

        sigsetmask (kernel_mask);               // Set interrupt mask for
                                                // the dispatcher.
        while (mainpgm -> state != T_DEAD) {
                register thread *go;            // Thread to run
                if (go = readyq -> next ()) {
                        go -> disp_proc();      // Dispatch to ready thread.
                        }
                else sigpause (0);                      // Idle CPU.
                }

        // Clean up

        status = mainpgm -> termination_status;
        delete mainpgm;
        delete clock;
}       return status;
        }
```

```
/*
 * $Header: pile.c,v 1.1 86/09/30 17:53:42 kenny Exp $
 *
 *      Definitions for the `pile' container type.
 *
 *      A ``pile'' is an empty container, so its operators all cause trouble.
 *
 * $Log:        pile.c,v $
 * Revision 1.1  86/09/30  17:53:42  kenny
 * Initial revision
 *
 */

#include "pile.h"

void empty_container (char *);

pile& pile::operator << (void * item) {
        register void * foo = item;             // Silence the warning
        empty_container ("pile::operator<<");
        return *this;
        }

void * pile::next () {
        empty_container ("pile::next");
        return 0;
        }

pile& pile::clear () {
        empty_container ("pile::clear");
        return *this;
        }

pile& pile::operator = (pile& p) {
        register pile *foo = &p;                // Silence the warning
        empty_container (
                "pile::operator=, assignment among piles is undefined");
        return *this;
        }

pile::pile (pile& p) {
        register pile *foo = &p;                // Silence the warning
        empty_container (
                "pile::pile, assignment among piles is undefined");
        }
```



```
/* $Header: pile.h,v 1.1 86/09/30 17:55:06 kenny Exp $
 *
 *      Declaration of the `pile' data type.
 *
 *      A `pile' is a set with two basic operations:
 *              - Add an object to the set.
 *              - Remove the first object from the set and return it.
 *      The order in which the objects will be returned by the `add' and
 *      `remove' operations is unspecified.
 *
 *      The `pile' is just a countainer class; derived classes of piles
 *      include stacks, queues, and priority queues.
 *
 * $Log:        pile.h,v $
 * Revision 1.1  86/09/30  17:55:06  kenny
 * Initial revision
 *
 */

#ifndef PILEH

class pile {
public:
        virtual pile& operator << (void *);     // Add an item
        virtual void * next ();                 // Remove & return an item
        virtual pile& clear ();                 // Empty the pile
        pile& operator = (pile&);               // Assignment among piles
        pile (pile&);                           // Copy in initialization
        pile () {}                              // Constructor
        };

#define PILEH
#endif PILEH
```

```
/*
 * $Header: queue.c,v 1.2 86/09/30 23:51:06 kenny Exp $
 *
 *      Implementation of linear queues.
 *
 *      A ``queue'' is a set of objects of arbitrary type, having enqueue
 *      and dequeue operations.  Enqueue adds an object to the queue, dequeue
 *      removes and returns the least recently enqueued object.
 *
 * $Log:        queue.c,v $
 * Revision 1.2  86/09/30  23:51:06  kenny
 * Added priority queues, implemented for now with linear search; a faster
 * version is desirable someday, at least for some applications.
 *
 * Revision 1.1  86/09/30  17:56:21  kenny
 * Initial revision
 *
 */

#include "queue.h"

const int NALL = 127;                   // Number of queue entries to
                                        // allocate at a time.

static queue_entry *qe_free = 0;        // Free space pool.

extern void abort ();

queue_entry::queue_entry (void * i, queue_entry * n) {  // Make a queue entry

        if (this) abort ();             // Cause an error if not new.

        if (!qe_free) {                         // If nothing in free pool,
                register queue_entry * p =      // get more space
                        (queue_entry *)
                                new char [NALL * sizeof (queue_entry)];
                for (int i=NALL-1; i>0; --i)
                        p[i].qe_next = p+i-1;
                p[0].qe_next = 0;
                qe_free = p + NALL - 1;
                }
        this = qe_free;                         // Get space for an entry
        qe_free = qe_next;

        qe_item = i;                            // Fill in the entry
        qe_next = n;
        }

queue_entry::~queue_entry () {                  // Destroy a queue entry
        qe_next = qe_free;
        qe_free = this;
        this = 0;
        }

pile& queue::operator << (void * item) {        // Add an item to a queue
        if (q_last == 0) {                      // Adding to empty queue?
```



```
                q_last = new queue_entry (item);
                q_last -> qe_next = q_last;
                }
        else {                                  // Adding to a non-empty queue
                q_last = q_last -> qe_next
                        = new queue_entry (item, q_last -> qe_next);
                }
        return *this;
        }

void * queue::next () {                         // Remove an item from a queue

        if (q_last == 0)                        // Removing from empty queue?
                return 0;

        register queue_entry *first = q_last -> qe_next;
                                                // Find the first item
        register void * item = first -> qe_item;

        if (first == q_last) {                  // Is queue now empty?
                q_last = 0;                     // Empty queue
                }
        else {
                q_last -> qe_next = first -> qe_next;   // Non-empty queue.
                }

        delete first;                           // Get rid of linkage node.
        return item;                            // Return first item
        }

pile& queue::clear () {                 // Clear out a queue
        register queue_entry *l = q_last;
        if (l != 0) {                           // Nothing to do if it's empty
                do {
                        register queue_entry *ll = l;
                        l = l -> qe_next;       // Delete all linkage nodes
                        delete ll;
                        } while (l != q_last);
                q_last = 0;
                }
        return *this;
        }

queue& queue::operator = (queue& q) {           // Assignment to a queue
        void empty_container (char *);
        queue *foo = &q;                        // Shut up the compiler.
        empty_container (
                "queue::operator=; assignment among queues is undefined");
        return *this;
        }

queue::queue () {                               // Queue constructor
        q_last = 0;                             // Initially empty
        }

queue::queue (queue& q) {                       // Copy in initialization
```

```
        void empty_container (char *);
        queue * foo = &q;                       // Shut up the compiler.
        empty_container (
                "queue::queue; assignment among queues is undefined");
        }

queue::~queue () {
        clear ();                       // Queue must be cleared before it's
        }                               //      destroyed.

pile& pqueue::operator << (void * item) {       // Add item to priority queue.
        register queue_entry *p = q_last;
        if (p == 0) {                   // Adding to empty queue?
                q_last = new queue_entry (item, 0);
                q_last -> qe_next = q_last;
                return *this;
                }
        register queue_entry *q = p -> qe_next;
        while ((*c) (item, q -> qe_item) >= 0) { // Search for spot to add
                p = q;
                q = p -> qe_next;
                if (p == q_last) break;
                }
        register queue_entry *r = new queue_entry (item, q);
        p -> qe_next = r;                       // Link item into queue
        if ((*c) (item, q -> qe_item) > 0)
                q_last = r;                     // Update tail if necessary
        return *this;
        }

pqueue::pqueue (comparator cc) : () {
        c = cc,                         // Initialize comparator pointer
        }
```



```
/*
 * $Header: queue.h,v 1.2 86/09/30 23:52:16 kenny Exp $
 *
 *      A ``queue'' is a set of objects with enqueue and dequeue operations.
 *      Enqueue places an object at the tail of the queue, and dequeue removes
 *      and returns the least recently enqueued object.
 *
 *      A ``priority queue'' is a queue with a defined comparison operation.
 *      Dequeue, instead of returning the least recently enqueued object,
 *      returns the first object in the ordering specified by the comparison
 *
 * $Log:        queue.h,v $
 * Revision 1.2  86/09/30  23:52:16  kenny
 * Added definitions for priority queues.
 *
 * Revision 1.1  86/09/30  17:56:46  kenny
 * Initial revision
 *
 */

#ifndef QUEUEH
#include "pile.h"

typedef int (*comparator) (void*, void*);

class queue_entry {
        friend class queue;
        friend class pqueue;

        queue_entry * qe_next;
        void * qe_item;

        queue_entry (void * i, queue_entry * n = 0);
        ~queue_entry ();
        };

class queue : public pile {
        friend class pqueue;

        queue_entry * q_last;
public:
        pile& operator << (void *);             // Add an item
        void * next ();                         // Remove and return an item
        pile& clear ();                         // Clear the queue
        queue& operator = (queue&);             // Assignment operator
        queue ();                               // Constructor
        queue (queue&);                         // Copy in assignment
        ~queue ();                              // Destructor
        };

class pqueue : public queue {
        comparator c;                           // Comparison function
public:
        pile& operator << (void *);             // Add to queue
        pqueue (comparator);                    // Destructor
        pqueue& operator = (pqueue&); {         // Assignment
```

```
        queue::operator = (q);
        return *this;
        }
    };

#define QUEUEH
#endif QUEUEH
```



```
/*
 * $Header: semaphore.c,v 1.2 86/10/08 15:42:24 kenny Exp $
 *
 *      Procedures to implement semaphores.
 *
 *      A semaphore is an abstract type with two operations, P() and V().
 * It may also be initialized and examined as if it were an integer.
 *
 *      Reference: Dijkstra, E. W. Hierarchial ordering of sequential
 *              processes.  Acta Informatica 1 (1971) 115-138.
 *
 * $Log:        semaphore.c,v $
 * Revision 1.2  86/10/08  15:42:24  kenny
 * Fixed off-by-one error on semaphore counts
 *
 * Revision 1.1  86/10/07  15:03:33  kenny
 * Initial revision
 *
 */

#include "hardware.h"
#include "queue.h"
#include "thread.h"
#include "semaphore.h"
#include "dispatch.h"
#include <signal.h>

//      UNIX kernel functions

int sigsetmask (int);                   // Set signal mask

semaphore& P (semaphore& s) {           // Wait on a semaphore
        register int sm = sigsetmask (kernel_mask);
                                        // BEGIN CRITICAL REGION
        if (--s.count < 0) {            // Decrement semaphore count
                _disp_end (D_ENQ, s.waiting);   // Block if negative
                }                       //   (and end critical region)
        else (void) sigsetmask (sm);    // END CRITICAL REGION
        return s;
        }

semaphore& V (semaphore& s) {           // Post to a semaphore
        register int sm = sigsetmask (kernel_mask);
                                        // BEGIN CRITICAL REGION
        if (++s.count < 1) {            // Increment semaphore count
                register thread *waiter = s.waiting -> next();
                                        // Get waiting thread
                if (waiter) waiter -> ready();  // Unblock waiting thread
                }
        (void) sigsetmask (sm);         // END CRITICAL REGION
        return s;
        }

semaphore::semaphore () {               // Initialize a semaphore
        waiting = new qptr(thread) (&q);
        count = 0;
```

```
        }

semaphore::semaphore (int initial) {
        waiting = new gpile(thread) (&q);
        count = initial;
        }

semaphore::~semaphore () {              // Destroy a semaphore
        delete waiting;
        }

semaphore::semaphore (semaphore& s) {   // Copy a semaphore - impossible
        semaphore *foo=&s;
        void empty_container (char *);
        empty_container ("semaphore::semaphore; can't copy semaphores");
        }

semaphore& semaphore::operator = (semaphore& s) {
        semaphore *foo=&s;
        void empty_container (char *);
        empty_container ("semaphore::operator=; can't copy semaphores");
        return *this;
        }
```



```
/* $Header: semaphore.h,v 1.1 86/10/07 15:03:52 kenny Exp $
 *
 *      Header file defining the semaphore data type.
 *
 *      A semaphore is an abstract type with two operations, P() and V().
 * It may also be initialized and examined as if it were an integer.
 *
 *      Reference: Dijkstra, E. W. Hierarchial ordering of sequential
 *              processes.  Acta Informatica 1 (1971) 115-138.
 *
 * $Log:        semaphore.h,v $
 * Revision 1.1  86/10/07  15:03:52  kenny
 * Initial revision
 *
 */

#ifndef SEMAPHOREH

#ifndef GPILEH
#include "gpile.h"
#endif
#ifndef QUEUEH
#include "queue.h"
#endif
#ifndef THREADH
class thread;                           // Make sure that the `thread' class
class gpile(thread);                    //      is defined.
#endif THREADH

class semaphore {
        friend semaphore& P(semaphore&);
        friend semaphore& V(semaphore&);
        int count;                      // Semaphore counter.
        queue q;
        gpile(thread) *waiting;         // Queue of threads blocked at
                                        //      semaphore
public:
        operator int () { return count; }       // Examine count
        semaphore& operator = (semaphore&);     // Copy semaphores
        semaphore (semaphore&);
        semaphore (int);                // Initialize semaphore
        semaphore ();
        ~semaphore ();                  // Destroy semaphore
        };

semaphore& P (semaphore& s);
semaphore& V (semaphore& s);
#define SEMAPHOREH
#endif SEMAPHOREH
```

```
/*
 * $Header: thread.c,v 1.2 86/10/08 00:58:54 kenny Exp $
 *
 *      Implementation of the ``thread'' primitive.
 *
 *      A ``thread'' is a lightweight process.  It has its own machine
 * conditions, its own flow of control, and its own stack.  It may have local
 * storage, if so, it is implemented with a derived class from the ``thread''
 * base class.
 *
 * $Log:        thread.c,v $
 * Revision 1.2  86/10/08  00:58:54  kenny
 * - Changed code so that current_thread is always 0 when executing in the
 *   dispatcher.
 * - Changed code so that sim_time is queued on the ready queue with the
 *   correct priority.
 *
 * Revision 1.1  86/10/07  15:05:29  kenny
 * Initial revision
 *
 */

#include <signal.h>
#include "hardware.h"
#include "queue.h"
#include "gpile.h"
#include "thread.h"
#include "semaphore.h"
#include "dispatch.h"
#include "interrupt.h"

implement(gpile,thread)

// UNIX kernel interface

void abort ();                          // Quit process, painfully.
int sigsetmask (int);                   // Set signal mask

// Timer interface

int cpuclock ();                        // Milliseconds since midnight

// User interface

void thread::ready () {                 // Make a thread ready to run
        state = T_READY;                        // Mark thread ``ready''.
        qpriority = priority;                   // Copy thread priority for
                                                //      queue sequencing.
        register int sm = sigsetmask (kernel_mask);
                                                // BEGIN CRITICAL SECTION
        *readyq << this;                        // Place thread on ready queue
        sigsetmask (sm);                        // END CRITICAL SECTION
        }

void thread::disp_proc () {             // Dispatch to a thread
        int argc;                       // Count of arguments to end dispatch
```



```
        int *argv;

        current_thread = this;          // Mark us the current thread
        state = T_RUNNING;              // Mark us running.
        register long start_time = cpuclock();  // Record our start time
        mc.dispatch (argc, argv);       // Dispatch to the thread
        veputime += (cpuclock() - start_time);  // Record CPU time used.
        current_thread = 0;

        if (argc == 0) abort ();                // Make sure that there are arguments
        switch (*argv++) {              // Select kernel function

                case D_END: {           // End of thread.
                        if (argc != 2) abort(); // Check arg count
                        state = T_DEAD; // End of thread. Mark it ``dead''.
                        termination_status = *argv++;
                                        // Record exit status
                        V (wfs);        // Unblock parent if waiting.
                        break; }

                case D_INTERRUPT: {     // Interrupt detected.
                        if (argc != 3) abort ();        // Check argument count
                        ready ();       // Return the task to the ready queue.
                        register int signo = *argv++;   // Get signal#
                        register int code = *argv++;            // Get code
                        catch_interrupt (signo, code);  // Field the interrupt
                        break; }

                case D_ENQ: {           // Block on some queue
                        if (argc != 2) abort ();        // Check argument count
                        state = T_BLOCKED;      // Mark the task blocked.
                        qpriority = priority;   // Update priority in case we
                                                // are blocking on the ready
                                                // queue (sim_time does this)
                        *((gpile(thread)*) *argv++) << this;
                                                // Enqueue the task.
                        break; }

                default:
                        abort ();

                }
        // Back to the dispatcher to fire up the next ready task.
        }

thread::thread (                // Create a new thread
        entryfunct what,                // Function to run as ``main program''
        int argc,                       // Number of arguments being passed
        void *argv,                     // Argument vector
        int prio,                       // Priority
        int stack                       // Stack size
                ) :     mc (what, argc, argv, stack),
                                        // Initialize machine conditions
                        wfs (0)         // Initialize ``wait-for-son'' semaphore
{
        priority = prio;                // Initialize priority
```

```
        veputime = 0;                   // Clear CPU time used
        wakeup_time = 0;                // Clear wakeup time
        register int sm = sigsetmask (kernel_mask);
                                        // BEGIN CRITICAL SECTION
        next_thread = first_thread;     // Link onto ``all threads'' list
        first_thread = this;
        ready ();                       // Make the thread ready.
        (void) sigsetmask (sm);         // END CRITICAL SECTION
        }

thread::~thread () {                    // Destroy a thread
        if (current_thread) P (wfs);    // If we're not destroying from kernel
                                        //      then wait for the thread to
                                        //      finish.
        state = T_BURIED;               // Thread is dead and buried.
        this = 0;                       // TEMPORARY KLUDGE
        }

thread::thread (thread& t) {
        thread *foo = &t;
        empty_container ("thread::thread; can't copy thread structures");
        }

thread& thread::operator = (thread& t) {
        thread *foo = &t;
        empty_container ("thread::operator=; can't copy thread structures");
        return *this;
        }
```



```
/*
 * $Header: thread.h,v 1.1 86/10/07 15:05:52 kenny Exp $
 *
 *      Header file for the ``thread'' primitive.
 *
 *      A ``thread'' is a lightweight process.  It has its own machine
 * conditions, its own flow of control, and its own stack.  It may have local
 * storage, if so, it is implemented with a derived class from the ``thread''
 * base class.
 *
 * $Log:        thread.h,v $
 * Revision 1.1  86/10/07  15:05:52  kenny
 * Initial revision
 *
 */

#ifndef THREADH

#ifndef GPILEH
#include "gpile.h"
#endif GPILEH
#ifndef HARDWAREH
#include "hardware.h"
#endif
#ifndef SEMAPHOREH
#include "semaphore.h"
#endif

typedef enum {                          // State of a thread
        T_READY,                                // On the ready queue
        T_RUNNING,                              // In execution
        T_BLOCKED,                              // Blocked
        T_DEAD,                                 // Terminated
        T_BURIED                                // Terminated and destroyed
        } thread_state;

class thread {

        thread * next_thread;           // Next thread on the list of all
                                        //  threads.
        Machine_Cond mc;                // Machine conditions
        semaphore wfs;                  // Wait-for-son semaphore
public:
        thread_state state;             // Current state of the thread
        long veputime;                  // Virtual CPU time consumed so far
        int priority;                   // Current priority
        int qpriority;                  // Copy of current priority. When a
                                        //      task is on the ready queue,
                                        //      priority may be changed,
                                        //      the ready queue is therefore
                                        //      sorted by qpriority so that
                                        //      it is never out of sequence.
        long wakeup_time;               // Time to wake up if task is delayed.
        int termination_status;         // Status with which a DEAD thread
                                        //      died.
```

```
        void ready();                   // Place task on ready queue.
        void disp_proc();               // Dispatch task and process kernel
                                        //      service request.

        thread (entryfunct, int, void*, int=0, int=0);
                                        // Constructor accepts entry function,
                                        //      argument count,
                                        //      argument list pointer,
                                        //      thread priority,
                                        // and  stack size.

        thread (thread&);               // Copy in initialization.
        thread& operator = (thread&);   // Assignment among threads.
        ~thread ();                     // Destructor does wait-for-sons.
        };

declare(gpile,thread)                   // There are piles of threads.

extern gpile(thread) *readyq;           // Ready queue of threads waiting
                                        //      to run.
extern gpile(thread) *delayq;           // Delay queue of threads waiting
                                        //      for a specified time.
extern thread *first_thread;            // First on list of all known threads.

extern thread *current_thread;          // Current thread if any.
#define THREADH
#endif
```



```
/*
 * $Header: time.c,v 1.3 86/10/08 15:43:46 kenny Exp $
 *
 *              Time management for NULLIX kernel.
 *
 *      This file contains two alternative time managers for the NULLIX
 * system kernel; they are ``sim_time'' and ``real_time'', respectively.
 * One or the other is started by the initialization routine in the
 * dispatcher, based on the value of ``sim_time_flag''.  Each runs as an
 * independent thread.
 *
 *      Simulated time operation is accomplished by maintaining a global
 * variable, ``tick.''  When there is no thread to dispatch, the timer thread
 * (which has the lowest priority of any thread on the ready queue) is
 * dispatched.  It advances ``tick'' to the wakeup time of the next thread
 * on the delay queue, unblocks that thread and any additional threads with
 * the same wakeup time, and returns to the ready queue to await the next
 * time that the processor goes idle.  If the delay queue is empty, it
 * executes a sigpause to await some external condition which will possibly
 * ready a process.
 *
 * $Log:        time.c,v $
 * Revision 1.3  86/10/08  15:43:46  kenny
 * Fixed bug in delay () routine for simulated time (wallclock should have
 * been clock).
 *
 * Revision 1.2  86/10/08  01:05:20  kenny
 * - Fixed incorrect call to setitimer.
 * - Forced sim_time to end dispatch after setting priority to give user_main
 *   a chance to start.
 * - Changed sigsem to vector of pointers to semaphores.
 * - Correct sign error in wakeup check in real_time.
 * - Added 'clock', 'await' and 'delay' functions.
 *
 * Revision 1.1  86/10/07  15:06:37  kenny
 * Initial revision
 *
 */

#include <signal.h>
#include <sys/time.h>
#include <sys/resource.h>
#include "hardware.h"
#include "queue.h"
#include "thread.h"
#include "semaphore.h"
#include "dispatch.h"
#include "interrupt.h"

// Unix system interface

void getrusage (int, struct rusage *);  // Get resource usage for process
void gettimeofday (struct timeval *, struct timezone *);
                                        // Get time of day
void setitimer (int, struct itimerval *, struct itimerval *);
                                        // Set an interval timer
```

```
void sigpause (int);                    // Pause and await a signal.
int sigsetmask (int);                   // Set the signal mask.

// Semaphore to protect the delay queue and timer structures.

static semaphore *timer_mutex;

// First task may be dequeued from delay queue before it's time to wake it.
// If so, it's recorded here.

static thread *next_wakeup = 0;

// Variable which holds time (real or simulated) of last wakeup.

long tick;

// Simulated time flag from initialization.

extern int sim_time_flag;

// VAX-specific clock primitives.

static long millisec (struct timeval tick) {
                                        // Convert timeval to milliseconds
        return (1000L*tick.tv_sec + tick.tv_usec/1000L);
        }

long cpuclock () {                      // Find virtual CPU time
        struct rusage amounts;
        getrusage (RUSAGE_SELF, &amounts);
        return millisec(amounts.ru_utime);
        }

long wallclock () {                     // Find real time-of-day
        struct timeval tick;
        gettimeofday (&tick, 0);
        return (millisec(tick));
        }

void set_alarm (long tick) {            // Set clock interrupt
        struct itimerval tock, rubbish;
        tock.it_interval.tv_sec = 0;
        tock.it_interval.tv_usec = 0;
        tock.it_value.tv_sec = tick / 1000L;
        tock.it_value.tv_usec = 1000L * (tick % 1000L);
        setitimer (ITIMER_REAL, &tock, &rubbish);
        }

// Simulated clock process.

int sim_time () {

        timer_mutex = new semaphore (1);        // Initialize semaphore

        current->real_priority = 8;     // Make sure that we will be
                                        // placed at the end of the ready
```



```
                                        // queue.

        _disp_end (D_ENQ, readyq);      // End dispatch; next dispatch
                                        // will indicate that system is idle.

        for (;;) {
                P(*timer_mutex);        // Begin critical region
                if (next_wakeup == 0) next_wakeup = delayq -> next ();
                                        // Get next task to awaken.
                if (next_wakeup) {
                        tick = next_wakeup -> wakeup_time;
                                        // Advance simulated clock
                        while (next_wakeup
                            && next_wakeup -> wakeup_time <= tick) {
                                next_wakeup -> ready();
                                        // Ready all tasks now due to wake up.
                                next_wakeup = delayq -> next ();
                                }
                        V(*timer_mutex);        // End critical section
                        }
                else {
                        V(*timer_mutex);        // End critical section
                        if (kernel_mask)
                                sigpause (user_mask);   // Idle processor
                        else return -1;                 // Deadlock
                        }
                _disp_end (D_ENQ, readyq);      // Return this task to ready
                                                // queue for next time.
                }
        /* NOTREACHED */
        }

// Real-time thread.
//      This function contains no ``return'' statement, because it is an
//      (intentionally) infinite loop.  It is declared as void, even though
//      it is used as int (), because otherwise the compiler will complain
//      that it returns no value.

void /*int*/ real_time () {

        timer_mutex = new semaphore (1);        // Initialize the semaphore

        register_interrupt (SIGALRM);           // Indicate that we wish to
                                                // handle alarm clocks

        for (;;) {
                P (*sigsem [SIGALRM]);          // Wait for an alarm clock
                                                // signal.
                P (*timer_mutex);               // Begin critical section.
                if (next_wakeup == 0)
                        next_wakeup = delayq -> next ();
                                                // Get process to awaken.
                while (next_wakeup
                    && (wallclock () >= next_wakeup -> wakeup_time)) {
                        next_wakeup -> ready ();        // Ready all wakeups
                        next_wakeup = delayq -> next ();
                        }
```

```
                if (next_wakeup) {
                        set_alarm (next_wakeup -> wakeup_time - wallclock ());
                                        // Set timer for next wakeup.
                        }
                tick = wallclock ();    // Record the current time.
                V (*timer_mutex);       // End critical section
                }
        /* NOTREACHED */
        }

//      Procedures for thread's interface to timers.

long clock () {                         // Absolute time.
        if (sim_time_flag) return (tick);
        else return (wallclock ());
        }

void await (long time) {                        // Wait for a specific time
        current_thread -> wakeup_time = time;   // Mark our wakeup time.
        if (time < clock ()) return;    // It's before the present, wake up.
        P(*timer_mutex);                // BEGIN CRITICAL REGION
        if (next_wakeup && next_wakeup -> wakeup_time > time) {
                                        // If there's already an alarm set,
                                        // and this process is waking up
                                        // sooner, then we need to fiddle:
                *delayq << next_wakeup; // Place the next wakeup back in the
                                        // queue.
                next_wakeup = 0;
                }
        if (!sim_time_flag) V(*sigsem [SIGALRM]);
                                        // Simulate a timer interrupt to force
                                        // real_time to reset the clock.
        sigsetmask (kernel_mask);       // Make sure that real_time can't
                                        // come in until this process joins
                                        // the delay queue.
        V(*timer_mutex);                // END CRITICAL REGION
        _disp_end (D_ENQ, delayq);      // Delay this process until the alarm.
        }

void delay (long time) {                // Delay for a specified time
        await (clock() + time);
        }
```